André Inácio Reis · Rolf Drechsler

Editors

Advanced Logic Synthesis

Springer

Advanced Logic Synthesis

André Inácio Reis • Rolf Drechsler
Editors

Advanced Logic Synthesis

Springer

Editors

André Inácio Reis
PPGC/PGMICRO
Institute of Informatics
UFRGS
Porto Alegre, RS, Brazil

Rolf Drechsler
Group for Computer Architecture
University of Bremen
Bremen, Germany

Cyber-Physical Systems
DFKI GmbH
Bremen, Germany

ISBN 978-3-319-67294-6 ISBN 978-3-319-67295-3 (eBook)
https://doi.org/10.1007/978-3-319-67295-3

Library of Congress Control Number: 2017956712

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

For more than four decades, the complexity of circuits and systems has grown according to Moore's law resulting in chips of several billion components. While already the synthesis on the different levels from the initial specification down to the layout is a challenging task, the quality of initial logic synthesis steps is still very determinant for the quality of the final circuit.

Logic synthesis has been evolving into new research directions, including the use of large–scale computing power available through data centers. The availability of warehouse computing opens the way to the use of big data analytics and cognitive applications from recent advances in artificial intelligence and infrastructure for parallel processing of graph data structures.

Besides these advances in computer science, the underlying fabrication technology evolution is also bringing emergent circuit technologies that require new synthesis techniques. Advanced fabrication nodes also require a tighter integration of logic synthesis and physical design, in order to bring physical awareness early in the design flow to achieve design convergence.

Logic synthesis can also be expanded to exploit higher level of abstractions, including the identification and manipulation of datapaths. This can be done by identifying data paths at the gate level, as well as by performing architectural transformations at higher levels of abstractions.

The field of SAT solvers and quantified Boolean formula (QBF) solvers has had recent important advances. This way, it is natural that several logic synthesis problems are being addressed, modeled, and solved with these tools.

Stochastic and statistical methods are also gaining importance in the field of logic synthesis. These include stochastic circuits that may arise in novel technologies, as well as methods of synthesis using probabilistic approaches.

This book celebrates the 25th edition of the International Workshop on Logic and Synthesis, which happened in June 2016. In doing so, we present a selection of chapters that originated as keynotes, special sessions, and regular papers from IWLS 2015 and IWLS 2016. The selection of papers reflects relevant topics for the advancement of logic synthesis. World–leading researchers contributed chapters, where they describe the underlying problems, possible solutions, and important directions for future work.

The chapters in the order as they appear in this book are:

- "EDA3.0: Implications to Logic Synthesis" by Leon Stok
- "Can Parallel Programming Revolutionize EDA Tools?" by Yi-Shan Lu and Keshav Pingali
- "Emerging Circuit Technologies: An Overview on the Next Generation of Circuits" by Robert Wille, Krishnendu Chakrabarty, Rolf Drechsler, and Priyank Kalla
- "Physical Awareness Starting at Technology-Independent Logic Synthesis" by André Reis and Jody Matos
- "Identifying Transparent Logic in Gate-Level Circuits" by Yu-Yun Dai and Robert K. Brayton
- "Automated Pipeline Transformations with Fluid Pipelines" by Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau
- "Analysis of Incomplete Circuits Using Dependency Quantified Boolean Formulas" by Ralf Wimmer, Karina Wimmer, Christoph Scholl, and Bernd Becker
- "Progressive Generation of Canonical Irredundant Sums of Products Using a SAT Solver" by Ana Petkovska, Alan Mishchenko, David Novo, Muhsen Owaida, and Paolo Ienne
- "A Branch-and-Bound-Based Minterm Assignment Algorithm for Synthesizing Stochastic Circuits" by Xuesong Peng and Weikang Qian
- "Decomposition of Index Generation Functions Using a Monte Carlo Method" by Tsutomo Sasao and Jon T. Butler

On the different abstraction layers, it is shown in which way logic synthesis can adapt to recent computer science and technological advances. The contributed chapters cover latest results in academia but also descriptions of industrial tools and users. Also, the chapters are very helpful in pointing out relevant novel topics for future work.

Porto Alegre, Brazil André Inácio Reis
Bremen, Germany Rolf Drechsler
June 2017

Acknowledgments

All contributions in this edited volume have been anonymously reviewed. We would like to express our thanks for the valuable comments of the reviewers and their fast feedback. Here, we also like to thank all the authors who did a great job in submitting contributions of very high quality. Our special thanks go to Jody Matos, Mayler Martins, Vinicius Callegaro, and Vinicius Possani for helping with the preparation of the book. Finally, we would like to thank Charles Glaser, Brian Halm, Nicole Lowary, and Murugesan Tamilselvan from Springer. All this would not have been possible without their steady support.

Contents

1 **EDA3.0: Implications to Logic Synthesis** 1
Leon Stok

2 **Can Parallel Programming Revolutionize EDA Tools?** 21
Yi-Shan Lu and Keshav Pingali

3 **Emerging Circuit Technologies: An Overview on the Next Generation of Circuits** 43
Robert Wille, Krishnendu Chakrabarty, Rolf Drechsler, and Priyank Kalla

4 **Physical Awareness Starting at Technology-Independent Logic Synthesis** 69
André Inácio Reis and Jody M.A. Matos

5 **Identifying Transparent Logic in Gate-Level Circuits** 103
Yu-Yun Dai and Robert K. Brayton

6 **Automated Pipeline Transformations with Fluid Pipelines** 125
Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau

7 **Analysis of Incomplete Circuits Using Dependency Quantified Boolean Formulas** 151
Ralf Wimmer, Karina Wimmer, Christoph Scholl, and Bernd Becker

8 **Progressive Generation of Canonical Irredundant Sums of Products Using a SAT Solver** 169
Ana Petkovska, Alan Mishchenko, David Novo, Muhsen Owaida, and Paolo Ienne

9 **A Branch-and-Bound-Based Minterm Assignment Algorithm for Synthesizing Stochastic Circuit** 189
Xuesong Peng and Weikang Qian

10 Decomposition of Index Generation Functions Using a Monte Carlo Method 209
Tsutomu Sasao and Jon T. Butler

Index 227

Chapter 1
EDA3.0: Implications to Logic Synthesis

Leon Stok

1 Introduction

Electronic Design Automation is ripe for a paradigm change. But is it ready? The ubiquitous availability of ample compute power will allow for a different way for designers to interact with their design data and will allow for new optimization algorithms to be invented. In this chapter we will introduce the concept of warehouse-scale computing [1], its software stack, and how it will apply to EDA analysis and optimization.

In the early days of Electronic Design Automation, EDA1.0, separate applications were developed. Each ran on individual workstations. Verification, synthesis, placement, and routing were separate tasks carried out by separate tools. The size of the designs that could be handled was limited by the available compute power and the scalability of the algorithms. But since Moore's law had not really taken off, these tools were sufficient for the design sizes at the time. The design flow was carried out by moving design data from tool to tool, with little interaction between the individual steps.

Due to Moore's law and the scaling of the technology designs became larger and larger. At the same time, it became more difficult to predict the result of decisions made early in the design flow on the final result. Wire delay started to play an increasingly larger role in the overall delay of a path because wire delays do not scale as well as gate delays over the last several technology generations. In addition, due to the larger design sizes a certain percentage of the wires became longer and longer. Logic synthesis therefore needed to really understand the effects of interconnect increasingly better to make sensible decisions. To make reasonable accurate timing predictions, placement and routing need to be fairly complete. In the

L. Stok (✉)
IBM Systems Group, 2455 South Road, Poughkeepsie, NY, 12601, USA
e-mail: leonstok@us.ibm.com

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_1

era of EDA2.0, this got handled by integrating the individual tools such as synthesis, placement, and routing as well as the analysis tools (such as timing) together in an integrated tool suite. These integrated tool suites would typically run on larger servers. Scaling to even larger design sizes was obtained by multi-threading many of the core algorithms such that they could take advantage of the many cores and threats in these larger servers.

In the last few generations, technology progress has slowed down from a power/performance perspective. Getting the utmost power and performance out of the smallest possible design has become more crucial to make a design in a new technology worth it. However, technology scaling has allowed for design sizes to continue to grow. At the same time design rule complexity has continued to go up, and people are advocating that handling complex rules needs to become an integral part of the design flow by providing in-design checking tools. As a result, the design work and possibilities for optimization have gone up tremendously. The amount of data that needs to be dealt with in an end-to-end design flow has exploded.

Design teams have increased in size. Despite that, it is impossible to complete a large design on an economically feasible schedule without lots of reuse. This has helped fill up the chips with homogeneous structures. But how many homogeneous cores do we want to continue to put on the same chip? The drive for better power/performance on specific workloads advocates for a lot more heterogeneity on a chip with functions that target a specific workload. To deliver this larger variety of designs, an additional boost in designer productivity will be required. It is time that we look beyond the individual, albeit integrated tools, and start to optimize the iterative end-to-end design flows.

Many of challenges for the future of EDA were outlined in a report on the 2009 NSF Workshop that focused on EDA's Past, Present, and Future. This report was published by Brayton and Cong in two parts in IEEE D&T [2]. The second part of that paper outlines the key EDA challenges. Interestingly, it has only a few challenges printed in bold: intuitive design environments, simplified user interfaces, standardized interfaces, and scalable design methodologies all leading to disciplined and predictable design. These challenges do not really drive the need for new EDA point tools. Instead they all point to problems that need to be solved in the end-to-end design flows. They point to the improvement that is needed in the design environment through which the designers interact with design tools and to the scale of problems that need to be solved.

These challenges have substantial overlap with the areas Big Data and Analytics applications have been focusing on and made tremendous progress in. One can certainly argue that a Big Data application like Google maps has a simple and intuitive user interface, has standard APIs to annotate data, and has been architected to be very scalable. It is therefore very pertinent to look how these applications have been architected and what that means for EDA3.0 applications.

It is time for the next era in EDA that attacks these problems. EDA3.0 will deliver this next step up in productivity. In this era, EDA needs to provide designers with analysis tools that do not just analyze the results and produce reports, but tools that provide real insight. The size of the reports has already become overwhelming for most designers. Analysis tools need to provide data that will give designers insight in how to make their design better. We need to move from the era of analysis tools to analytics tools. These tools should take advantage of the compute power of large warehouse-scale clusters instead of individual servers such that they can provide near real-time insight in the state of a design. At the same time, we want to harness the power of these large compute clusters to devise smarter optimization algorithms that explore a larger part of the design space.

What will need to happen to make EDA3.0 reality? First we will have to capitalize on the changing nature of IT. We need to learn from Big Data, Cognitive and other data, and graph parallel systems. This will allow us to create an integrated design flow on very large compute clusters. Next, we need to change the way designers interact with design data and allow them to get much better insight in the state of their design. They need to understand what needs to be done next to meet their constraints. The analytics tools need to provide this insight. Finally, a new class of optimization algorithms needs to be invented that deliver a faster convergence and therefore designer turn around time (TAT) in meeting the design objectives on increasingly larger designs.

2 Warehouse-Scale Computing

The term warehouse-scale computer was introduced in [1]. "The trend toward server-side computing and the exploding popularity of Internet services has created a new class of computing systems that we have named *warehouse-scale computers*, or *WSC*s. The name is meant to call attention to the most distinguishing feature of these machines: the massive scale of their software infrastructure, data repositories, and hardware platform. This perspective is a departure from a view of the computing problem that implicitly assumes a model where one program runs in a single machine. In warehouse-scale computing, the program is an Internet service, which may consist of tens or more individual programs that interact to implement complex end-user services such as email, search, or maps."

The authors point out many advantages of warehouse-scale computing. Many of them applicable to Electronic Design Automation as well. Before discussing the implications to EDA, let us look deeper into the Software Infrastructure needed for warehouse-scale computing. The authors define three typical software layers in a WSC deployment.

- *Platform-level software:* the common firmware, kernel, operating system distribution, and libraries expected to be present in all individual servers to abstract the hardware of a single machine and provide basic server-level services.
- *Cluster-level infrastructure:* the collection of distributed systems software that manages resources and provides services at the cluster level; ultimately, we consider these services as an operating system for a datacenter. Examples are distributed file systems, schedulers and remote procedure call (RPC) libraries, as

well as programming models that simplify the usage of resources at the scale of datacenters, such as MapReduce [3], Dryad [4], Hadoop [5], Sawzall [6], BigTable [7], Dynamo [8], Dremel [9], Spanner [10], and Chubby [11].

- *Application-level software:* software that implements a specific service. It is often useful to further divide application-level software into online services and offline computations because those tend to have different requirements. Examples of online services are Google search, Gmail, and Google Maps. Offline computations are typically used in large-scale data analysis or as part of the pipeline that generates the data used in online services; for example, building an index of the Web or processing satellite images to create map tiles for the online service.

How would this apply to a typical Electronic Design environment? Let us look at each of the three layers and start with the Platform-level software. In the early days of WSC, only large organizations like Google and Facebook were able to provide uniform platform-level software in their public clouds. But recent developments such as OpenStack [12] are making similar capabilities available to private clouds. While many EDA users have private clouds that are quite heterogeneous and different from enterprise to enterprise, technologies such as OpenStack drive to homogenize these heterogeneous environments. OpenStack software allows one to control large pools of compute, storage and networking resources throughout a datacenter, managed through a dashboard or via the OpenStack API. OpenStack works with popular enterprise and open source technologies making it ideal for heterogeneous infrastructure. This platform-level software is applicable to the EDA environment mostly as-is.

The cluster-level software provides the management of the resources and the programming models that simplify the usage of these resources. The management portion is in general applicable to EDA applications as well. For example, distributed file systems are key to store the huge amount of design data. Schedulers like Platform LSF [13] have been commonplace in EDA environments and the counter pieces in warehouse-scale computing are equally important.

Several of the programming models are relevant to EDA applications. For simplicity, let us divide the data in an electronic design flow in two categories: core-design data and derived data.

- *Core design data* is the data created by the designers such as their Verilog and VHDL descriptions, their floorplan, the timing assertions, the base IP blocks and libraries as well as the data added by construction tools like synthesis, placement and routing to enable tape-out of OASIS.
- *Derived data* is the data produced by (mostly) analysis tools such as: verification traces and coverage, timing, noise and power reports, placement and routing density reports, audit and coverage reports.

Since our chips have grown in size and the capabilities and speed of the analysis tools have drastically improved the amount of derived design data in a design process has exploded. It has become a huge challenge for a design team to manage

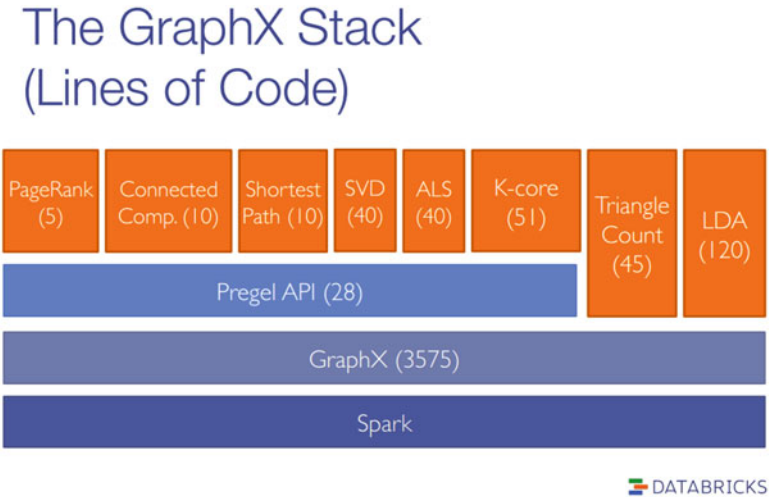


Fig. 1.1 The GraphX stack

this volume of data and more importantly make sense out of it. The insights needed to drive the next iteration of the design process have been more and more difficult to obtain. Many of the programming models described under the Cluster-level infrastructure are very well suited for this type of derived data. A programming paradigm like Bigtable [7] applies really well to most of the well-structured derived data that gets generated in the design process. Similarly, other non-SQL, SQL, and graph databases are very well suited for the volumes of derived data generated in a typical design process.

Cluster-level infrastructure for the core design data is less readily available. EDA research and development has spent many years optimizing core data-structures for fast logic simulation, quick synthesis and place and route, or huge scale checking for timing, LVS or DRC. However, time has come to relook at these now that we enter the era of warehouse-scale computing.

A programming model like Spark has become the basis for many algorithms that are very similar to core algorithms in EDA tools. For example, Fig. 1.1 shows the stack for the GraphX model built on top of Spark. It provides many components for key graph algorithms, such as shortest path algorithms, singular value decomposition, and connected component calculations.

What would be the key programming models that are needed in the EDA space? A distributed version of a network database like OpenAcces(OA), if it were to be designed to scale to very large sizes as Bigtable does, would be an essential programming model in the EDA stack. A very scalable implementation of a graph database would be a great foundation for analysis tools like Static Timing Analysis. Other key programming models are: a model to handle matrices for circuit simulators and a model for graphics data for layout applications. Fortunately, there are only a handful of these core programming models that can cover most of the EDA applications.

Creating these cluster-level programming models for EDA specific usage should be a very fertile ground for academic research. Many of the programming models used widely in analytics, big data and cloud applications are Open Source and came directly out of academic research. In addition to their availability under Open Source licenses, they formed the basis for a new generation of companies. Take, for example, Spark [15], which came out of UCB's AMPlab and led to the foundation of a company like DataBricks [16].

Once several of these cluster-level programming models are in place, EDA research and development should focus on the application-level software. To understand the application space, it is important to take a fresh look at how an end-to-end design flow should be built from these applications and not look at a one-to-one replacement of our current EDA tools. We need to look at the iterative design process through the same lens as many of the big data applications and split them in online services (e.g., interactive, fast response) and offline computations. Take, for example, static timing analysis. Performing the timing analysis itself can take a significant number of hours on a large chip (e.g., the offline computation). However, the user wants to be able to traverse the resulting outcome quickly to find specific problems and devise strategies to address them (e.g., the online service). The insights from the online service are crucial for the designer to drive the next iteration of the design process which can take place as the next offline computation to provide data for the next iteration of analysis. If we can extend this model such that we have a live database (like Google Maps) which reflects the latest state of the entire design (and has versioning to roll-back to earlier versions of the design), this model will be the basis for fast queries.

As soon as the offline computations are finished, they will be reflected back (and versioned) into the live model. This will allow designers to get quick insight in the current state of the entire design and its latest interactions between its partitions. In the next section, we will compare the scale of some of these analytics and big data applications with the scale of EDA applications, to understand if their programming models are suitable for the design sizes we typically encounter in EDA applications.

3 Scale of Applications

In this section we will look at the question if the programming models in the cluster-level infrastructure can scale to the level of EDA applications. We will look at various applications and compare them to typical EDA applications.

Most EDA tools are deployed in complex design flows and used by a large number of designers. Many tools communicate with each other through arcane file-formats [30] and produce data in huge text based log and report files. Most tools have a very large number of configuration parameters and require a very elaborate setup for a particular technology or design style, usually reflected in elaborate and complex control scripts.

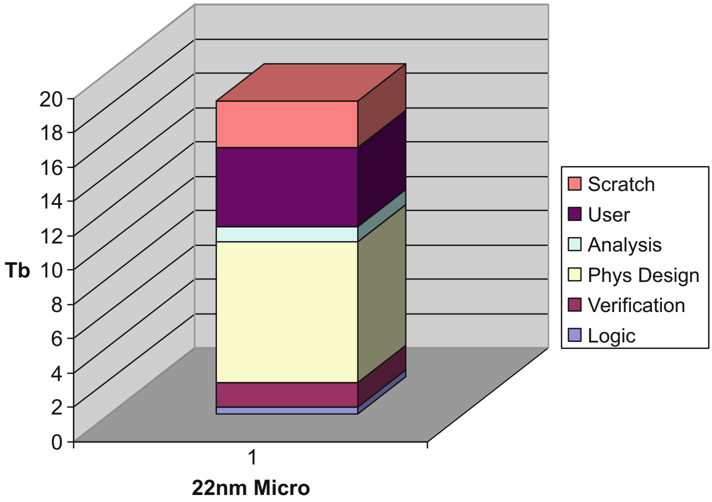


Fig. 1.2 Design data for a 10 + B transistor 22 nm chip

It is not uncommon for design teams to write 1–2 millions of lines of Skill or Python scripts to create library cells and IP blocks. A complete design and verification flow can quickly add up to 1–2 million lines of TCL to control the tools and deal with the setup and environment in which IP and models are stored. And once the design flow is up and running, a large number of Python and Perl scripts are written to extract the key information from the terabytes of reports and design data. It is often difficult to know how many of these scripts exist in a design environment since they are often owned by individual designers. In addition to all the configuration and control files a large amount of data gets generated and stored.

Let us look at the amount of data produced by a design team designing a 10B+ transistor processor in 22 nm technology as shown in Fig. 1.2. It takes about 12 Tb to store the entire golden data (including incremental revisions). Logic and verification setup takes about 2 Tb, the physical design data about 8–9 Tb and another 1–2 Tb for analysis reports. In addition, individual users keep another 6 Tb of local copies in user and scratch spaces. For functional verification, approximate 1.5 Tb of coverage data is collected daily. This data is very transient and about 2 weeks worth of data (21 Tb) is kept in a typical verification process.

After the design is finished, the physical data get compressed and streamed out to about 3Gb of OASIS. The product engineering team blows this up to around 1 Tb during mask preparation operations. Finally, another 5 Tb of test and diagnostics data gets generated in the post-silicon process.

This looks like a significant amount of data but it tops out at 50 Tb for a multi-year design and manufacturing project. Recently a study [17] was published that produced the table in Fig. 1.3. It seems to project the 22 nm chip storage needs

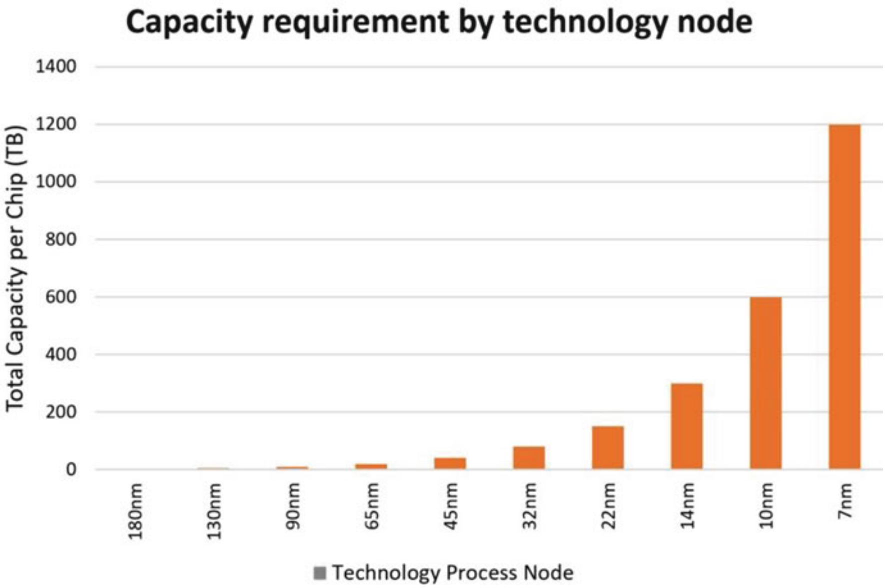


Fig. 1.3 Estimated storage capacity requirements by EDA tools for the entire RTL-to-GDSII flow per chip design versus technology process node. (Source: Dell EMC)

at a 150 Tb or about 3x our estimate. It also projects a 2x increase in each of the subsequent technology nodes.

While this was once a phenomenal amount of data that put EDA at the forefront of storage and compute needs, it has been surpassed by many large cloud-based Big Data applications. Let us look at some of them.

Let us try to compare the scale of some of the cloud applications with the problem set we deal with in EDA. We will take a look at Google Maps [18] and some key metrics that we can compare with the design data above.

1. *How much data has Google Maps accumulated?*
   Combining satellite, aerial and street level imagery, Google Maps has over 20 petabytes of data, which is equal to approximately 21 million GB, or around 20,500 Tb
2. *How often are the images updated?*
   Depending on data availability, aerial and satellite images are updated every 2 weeks. Street View images are updated as quickly as possible, though Google wasn't able to offer specific schedules, due to its dependence on factors such as weather, driving conditions, etc.
3. *In the history of Google Maps, how many Street View images have been taken?*
   The Street View team has taken tens of millions of images since the Street View project began in 2007, and they've driven more than 5 million unique miles of road.

How does this compare to the design data? Comparing the 50 Tb/design versus the 20 Pb of the fully annotated Google maps, the design data is only 1/400th the size. A large chip has 5 km of wire compared to the 5 million miles of road to accumulate street view images. Of course, the scale of intersections on the chips are in nm's and the road crossings are in kilometers. It would be interesting to compare the number of road intersections in the world with the number of vias on a chip. Typically, the street view data annotated with each street is significantly larger than the physical design data needed to be associated with each stretch of wire.

One major difference is that the core EDA design data is certainly more dynamic than the more static base map of roads in an application like Google maps. EDA tools can much more quickly reroute wires than physical roads can be built. But let us look at another data point to illustrate the velocity of data in a cloud application like YouTube. Each minute 300 h of video is uploaded to YouTube [19]. This is indexed, categorized, and made available. While we have no accurate data on how much of the design data changes each day, since it tops out at 50 Tb after a multi-year project, it is safe to assume that only a small fraction of it changes daily.

Based on these examples we conclude that many of the cluster-level programming models will be able to handle the typical data sizes in EDA projects. Let us look at the traversal speed of some of these models as well. Graph databases have become one of the fastest growing segments in the database industry. Graph databases not only perform well in a distributed environment but can also take advantage of accelerators such as GPUs. Blazegraph set up an experiment to run a Parallel Breadth First search on a cluster of GPUs. Using such a cluster, Blazegraph demonstrated a throughput of 32 Billion Traversed Edges Per Second (32 GTEPS), traversing a scale-free graph of 4.3 billion directed edges in 0.15 s [20]. This is a few orders of magnitudes more than a static timing analysis tool which traverses about 10 M edges per second on a single machine. An example of a matrix programming model is shown in Quadratic Programming Solver for Non-negative Matrix Factorization with Spark [21].

Despite the applicability of many of these programming models to EDA relatively little attention has been paid to them. This is caused by the fact that most of the public discussion has been overshadowed by other "cloud" aspects and specifically the element of data security [22, 23]. Indeed, only when sufficient security guarantees are given will designers put their entire IP portfolio on a public cloud. However, EDA applications can run in private (or hybrid) clouds and take full advantage of the massively distributed warehouse-scale computing infrastructure without the security issues.

Unfortunately, this heavy focus on the security aspect has overshadowed the discussion around the opportunities of warehouse-scale computing to the EDA design flows and applications. This is also the reason I am using the term "warehouse scale computing" instead of cloud to not distract from its underlying potential. In the next section, we will describe how EDA tools can take advantage of the warehouse-scale software infrastructure. We will describe how we can make a design flow a lot more productive and designer-friendly.

4 EDA Applications

What does a designer (the EDA tool client) really want? She wants to get to the design DATA from anywhere and any place. She wants the DATA to be there without her waiting for it. She wants to analyze the DATA with whatever tools she can lay her hands on to learn how to improve her design. She wants to know how to get from A to B through the design process and wants design data, design navigation, and a design flow to act like Google maps. For example, wouldn't it be great if understanding timing and congestion problems in your design is no more difficult than turning on traffic congestion information in Google maps? Wouldn't it be great if we could annotate key manufacturing data from inline inspection tools just as easy as Street Views to our design data? This has certainly become easier to accomplish using key elements from cluster level programming models.

This type of rapid analysis and optimization can only be accomplished if the entire design data is in a (set of) live database(s) distributed among many machines in the warehouse-scale compute center. When design changes are made, the live data needs to be incrementally communicated and updated. We know how to do this for timing analysis integrated within a place and route flow. However, this incrementality needs to be extended to all analysis. Analysis engines can run on many parts of the design simultaneously and can be folded together in the live model. The analysis tools will produce the appropriate abstractions that are needed by the higher levels of hierarchy in the design.

Instead of thinking about synthesis, place, route, and timing algorithms, this DATA-centric EDA3.0 paradigm will start from the data, map it to the compute infrastructure using the right cluster-level infrastructure, and put the applications (e.g., placement, routing, timing analysis) on top of that using well-defined cluster-level APIs as services.

Clearly there are some technical challenges here. EDA data is more connected than many of the social networking applications. However, in many applications we have seen that EDA data viewed the right way is inherently more parallel than initially thought. The fact that tens or hundreds designers can work simultaneously and productively on a design makes it clear that the parallelism exists in the design process, albeit sometimes not in a single optimization run.

While EDA data is certainly more volatile during the optimization part of the process, the increased re-use of IP and increased use of hierarchy with appropriate abstractions has resulted in a much larger portion of the design data to be stable in the iterations of modern hierarchical designs. Furthermore, only a small portion of the chip and logic design gets (re-)done each day. With advanced version control fully integrated in the data itself, the knowledge of what actually changed can lead to a whole new class of optimization algorithms.

Large service providers such as Google and Facebook provide their own platform and cluster-level infrastructure based on special versions of open source code. Many smaller companies will build on standardized platforms such as CloudFoundry [24] and Bluemix [25]. The EDA industry needs to ask itself the question: what can we do to customize several of the cluster-level programming models to allow EDA tools to be written like services that interface with live design data through well-defined APIs?

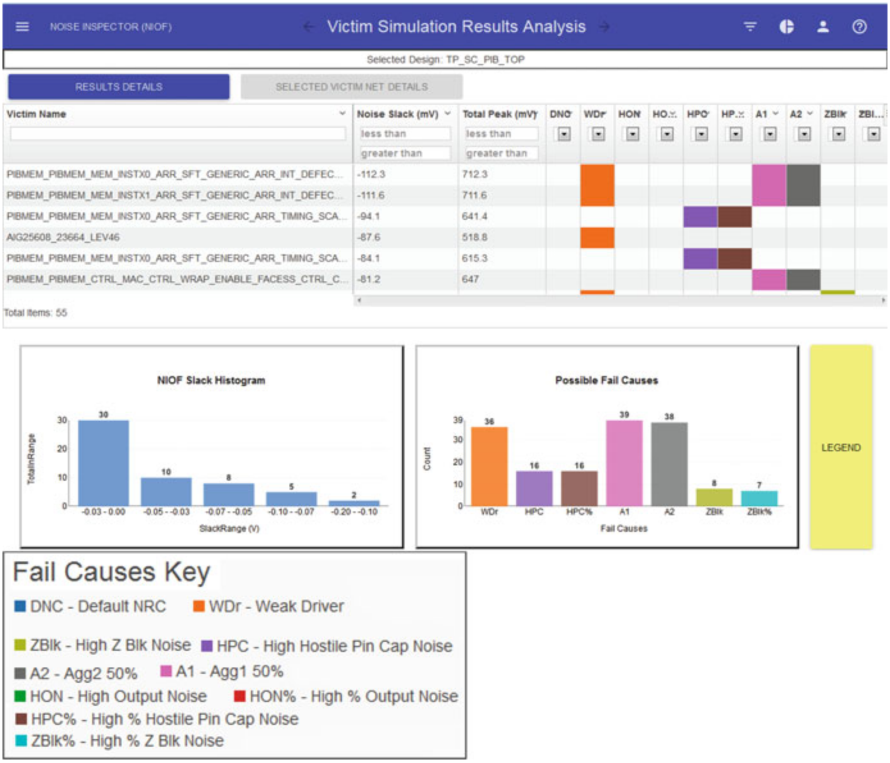
5 EDA Applications: Analysis

As described in Sect. 2 we can leverage many of the existing cluster-level infrastructure programming models for EDA applications. This is particularly true for the EDA applications that perform analysis and reporting. DRC, LVS, and timing analysis applications are well geared to the distributed infrastructure.

As an example, we deployed Neo4j [26] for timing analysis. Using Neo4j we now have an unprecedented capability to store detailed timing information in a highly efficient format, enabling very powerful query analysis. Our current benchmarking efforts have established the feasibility of storing a complete top-level chip timing graph from EinsTimer consisting of nearly 100 million nodes and associated timing properties in Neo4j. Using an IBM Power8 Linux server, we are able to import data within 11 minutes. The resulting graph database is indexed on several key query parameters, such instance names and slack values, within a matter of a few minutes. Once indexed, we have demonstrated the ability to perform lookups, for example iterating through all timing end points in the top-level chip run, and returning the worst N sorted by slack within 2 seconds. Given the underlying graph schema consisting of both properties and labels, we have the ability to support multiple levels of hierarchy within a single graph instance, which then enables efficient cross-hierarchy analysis within a single instance of Neo4J. For example, we have demonstrated the ability to compare in vs. out of context timing for approximately 1000 primary inputs on given macro within its top-level environment within 5 seconds. Previously, such cross-hierarchy analysis would require complex code to parse timing reports and resolve the names throughout the hierarchy. Such powerful analytics can often be achieved in a single line of Neo4J Cypher, illustrating the power of graph databases in analyzing complex EDA data sets.

The other advantage of using a graph database as a timing data server is that it can provide the data as an always on web service. We are fully leveraging Neo4j as a web-server which subsequently allows for a rich variety of applications to be built on top of the underlying graph database service, allowing us to package numerous queries as REST end points - as well as visualize results using custom built web-applications which allow for a clean visual representation of key timing metrics. From the standpoint of a chip designer, the above framework provides “any time anywhere” access to critical EDA timing results for use in triage, as well as trend analysis over time.

We use a similar framework for noise analysis. In Fig. 1.4 we show an example of a Noise inspection tool that we developed on top of an analytics framework. The cause of noise problems is often not very easy to determine. Noise can be caused by a combination of weak drivers, low tolerance for noise on sink gates, large aggressor



Selected Design: TP_SC_PIB_TOP

Victim Name	Noise Slack (mV)	Total Peak (mV)	DNC	WDr	HON	HO...	HPC	HP...	A1	A2	ZBlk	ZBl...
	less than greater than	less than greater than										
PIBMEM_PIBMEM_MEM_INSTX0_ARR_SFT_GENERIC_ARR_INT_DEFEC...	-112.3	712.3		■					■	■		
PIBMEM_PIBMEM_MEM_INSTX1_ARR_SFT_GENERIC_ARR_INT_DEFEC...	-111.6	711.6		■					■	■		
PIBMEM_PIBMEM_MEM_INSTX0_ARR_SFT_GENERIC_ARR_TIMING_SCA...	-94.1	641.4					■	■				
AIG25608_23664_LEV46	-87.6	518.8		■								
PIBMEM_PIBMEM_MEM_INSTX0_ARR_SFT_GENERIC_ARR_TIMING_SCA...	-84.1	615.3					■	■				
PIBMEM_PIBMEM_CTRL_MAC_CTRL_WRAP_ENABLE_FACESS_CTRL_C...	-81.2	647							■	■		

Total Items: 55

Fail Causes Key

- ■ DNC - Default NRC
- ■ WDr - Weak Driver
- ■ ZBlk - High Z Blk Noise
- ■ HPC - High Hostile Pin Cap Noise
- ■ A2 - Agg2 50%
- ■ A1 - Agg1 50%
- ■ HON - High Output Noise
- ■ HON% - High % Output Noise
- ■ HPC% - High % Hostile Pin Cap Noise
- ■ ZBlk% - High % Z Blk Noise

Fig. 1.4 Noise inspection example

couplings, DC aggressor nets, and various other reasons. For a designer or synthesis or routing tool to effectively combat noise problems it is important to get a good profile of the likelihood of each of these causes to try out the best remedies. This requires both a global analysis of the entire design to look for specific weaknesses in certain IP blocks and a detailed drill-down capability to address specific problems on specific instances in a design. We wrote the output of our EinsNoise tool in MongoDB. Using the "R" analysis language we were able to quickly classify these sources as shown in Fig. 1.4.

6 EDA Applications: Synthesis and Optimization

In Sect. 5 we described how several of the programming models apply to EDA analysis applications. A key question is how we can use the massive parallelism for optimization functions such as logic synthesis, placement, and routing. A lot of research and development has gone into multi-threaded applications for optimiza-

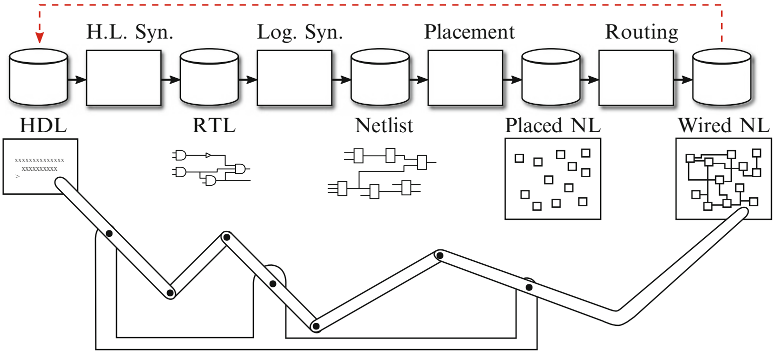
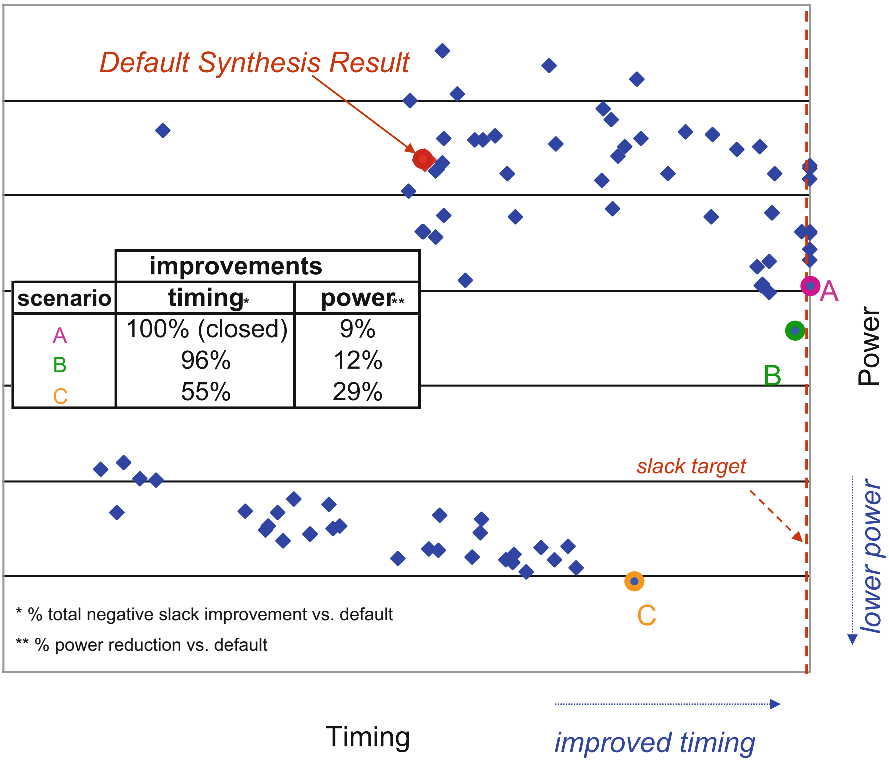


Fig. 1.5 Lack of predictability in the design flow

tion. However, effective strategies to leverage massive distributed infrastructure in optimization have not been widely considered. One of the reasons is that most research has focused on optimizing a single algorithm or application instead of looking across the entire design process. Let us look closer at the synthesis, place, and route flow.

In a design project, designers iterate numerous times across this flow. The main reason for these iterations is the lack of prediction of the effect of early decisions on the final outcome. This is best illustrated in the diagram in Fig. 1.5. When a designer is writing her HDL it is still very difficult to predict the outcome on the final placed and routed result. To improve the predictability, we include some prediction of placement into the logic synthesis steps and some routing prediction into the placement steps. But since these steps are fairly compute intensive only a very small part of the design space is explored. This can be illustrated by the effects that tool parameters have on the result. Designers affect the behavior of the tools by hints and the numerous parameters that each of the optimization tools have. Changing a few of these settings can lead to drastically different outcomes. The designers explore this opportunity to get their designs to meet power or timing objectives or ensure they are routable.

Let us look at an example in Fig. 1.6. This graph shows the results of the same design after synthesis, placement, and routing. The difference between each of the points on the graph is the different parameters that were used in the synthesis tools. The horizontal access shows the results with respect to timing. The vertical access shows the results with respect to power. Improved timing is toward the right on the x-axis. Improved power is lower on the y-axis. The result of the run with default parameters is the red diamond in the middle of the graph. The default result met the timing constraints but significant optimization to improve power is still possible. The blue diamonds show more than one hundred results after placement and routing. The only difference is the parameters used in synthesis. Of particular interest are points A, B, and C. Point A exactly meets timing, but uses 9% less power than the default run. Point C is of particular interest. While not as fast as the default, timing is still better than point A. Power is reduced by 30% versus the default run. This



	improvements	
scenario	timing*	power**
A	100% (closed)	9%
B	96%	12%
C	55%	29%

Fig. 1.6 Impact of synthesis parameters on final placed and routed designs

example is an illustration of the drastic effect that synthesis parameters can have on the final outcome of the optimization process.

In a practical hierarchical design process, the requirements on the implementation of a block change frequently throughout the process. Floorplans might need to be adjusted to accommodate other blocks. Many timing paths run across multiple blocks and timing assertions for one block will depend on the quality of results of other blocks that share these paths.

Running the tools end to end with just a few different parameters is quite wasteful. Most of the steps are repeated in each of the runs, and only the steps following the ones that are affected by the specific parameter changes will produce different results. Can we change the paradigm in optimization to take advantage of the cluster programming models that run very efficiently on warehouse-scale computers?

Decisions made during synthesis have a major impact on the outcome of synthesis itself and especially on the structure of the resulting netlist. While subsequent optimizations during physical synthesis will make modifications to this structure, these optimizations are mostly local and do not change the bias that was put in early in the process. Stok observed in the wavefront technology mapping paper [27] that technology mapping is the stage in logic synthesis where a most of the netlist structure gets locked in. By giving more alternatives to technology mapping, it is able to make better choices. As shown in Fig. 1.7, the wavefront paper introduced the concept of the multi-input (choice nets) which allows technology mapping to choose one of the inputs when determining the actual technology implementation of the gate. If timing optimization is the goal, technology mapping can choose the input that produces the fastest timing for each cone of logic (covering step) independently. This will potentially lead to an area increase since implicit cloning of gates is taking place, but can certainly help a lot to get to the fastest timing results.

Chatterjee [28] expands on this idea and shows how to reduce the structural bias in technology mapping by creating even more alternatives in the netlist before technology mapping, so the covering step has more alternatives to choose from. In Fig. 1.8 it is shown how the eliminate, simplify, factoring and re-substitution transformations are changed to add many choices to the netlist before technology mapping.

However, in both of these approaches the choices are locked in during technology mapping. The wavefront algorithm has the advantage that it can see the impact of all gates being mapped to a particular technology implementation before making technology choices in the second pass. While this certainly improved the choices that technology mapping makes, it still required these choices to be made with very imperfect information. We are making these choices before any of the impact of placement, routing or routing congestion is understood. In general, in the overall optimization process we perform many optimizations in sequence, try many alternatives, but quickly lock in a solution before we go into the next optimization step. In many cases, we make the choices before having adequate information to do so. As a result, synthesis algorithms make the major decisions on the structure of the netlist, which in turn will have very significant impact on placement and routing, without having adequate information on the consequences of its choices.

Instead, we would like to leave the “alternative” choices in the network until placement and routing have been completed and only resolve the best choices that make us meet our design constraints once we have sufficient reliable information to do so. However, at the time of creation of the wavefront algorithm we were limited by the available compute power and could not deal with the explosion in data that needs to be handled in placement, timing and routing if all the choices are left around. However, we are getting to a time where this is becoming practical. If we look at applications that run really well on warehouse-scale computers, they very effectively distribute the compute workload across many lighter weight computations.

Let us look at Watson as an example. The DeepQA [29] architecture is underlying the capability of Watson to understand and answer questions. Figure 1.9 shows a high-level picture of the DeepQA architecture. “The DeepQA architecture views the problem of Automatic Question Answering as a massively parallel hypothesis generation and evaluation task. As a result, DeepQA is not just about question-in/answer-out—rather it can be viewed as a system that performs differential

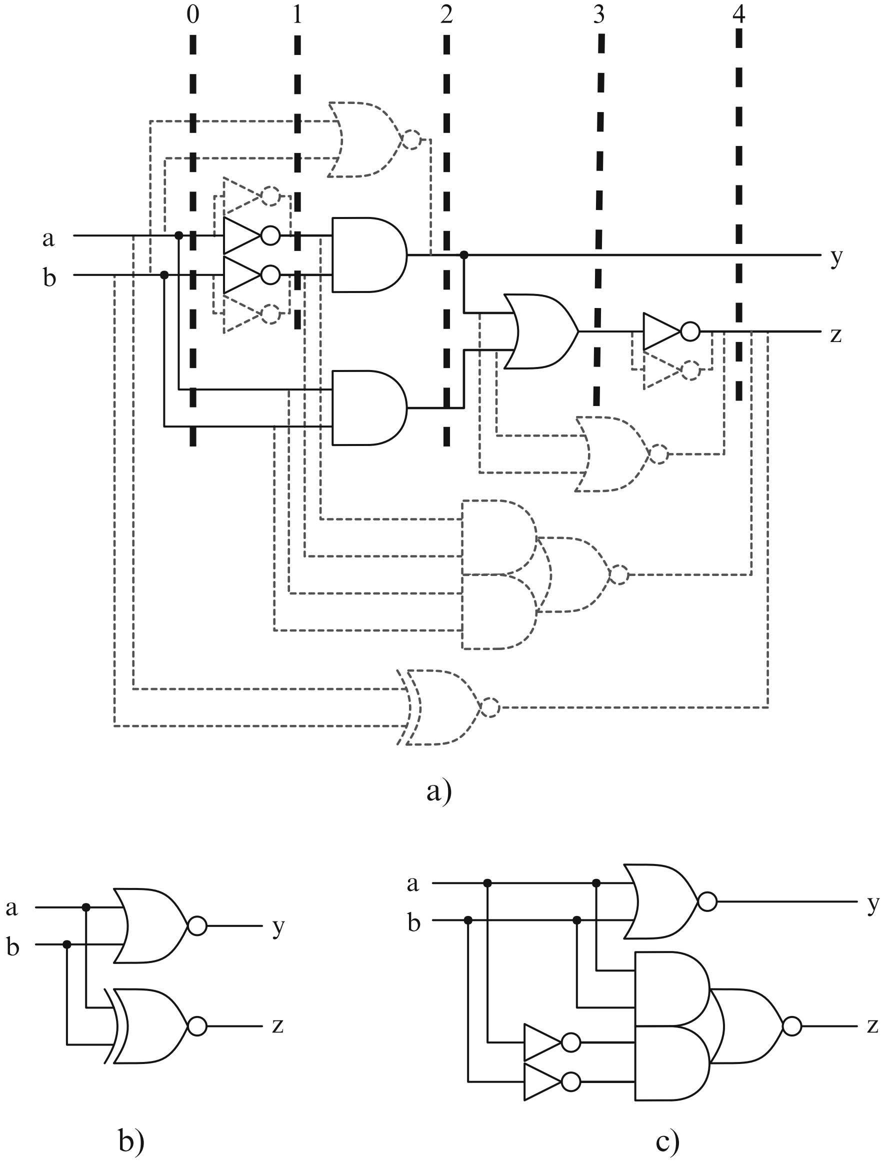


Fig. 1.7 Wavefront technology mapping, L. Stok, M. Iyer, A.J. Sullivan, 1998 IWLS

diagnosis: it generates a wide range of possibilities and for each develops a level of confidence by gathering, analyzing and assessing evidence based on available data."

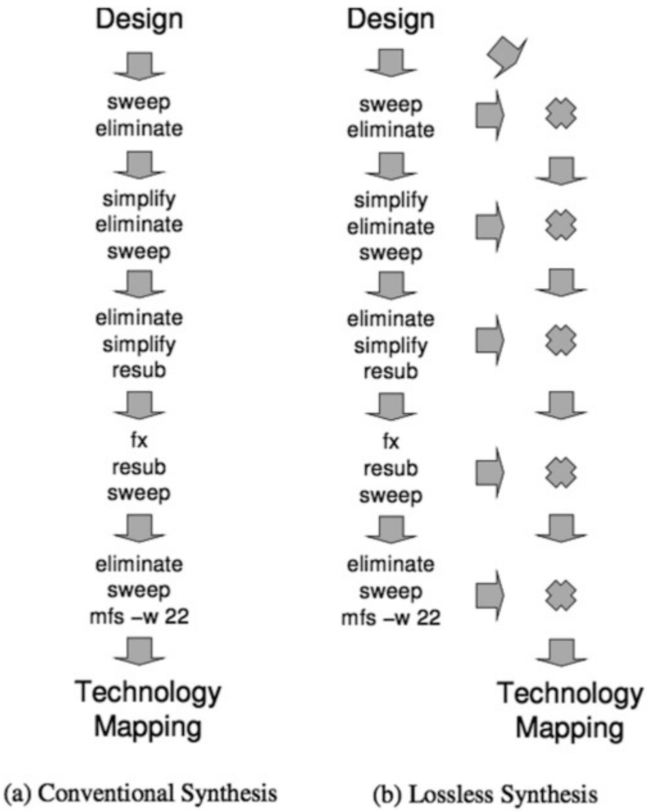


Fig. 1.8 Reducing Structural Bias in Technology Mapping, S. Chatterjee A. Mishchenko R. Brayton, X. Wang T. Kam, 2005 TCAD

What if we looked at the synthesis, placement, and routing design flow through a similar lens? Logic synthesis will be the candidate generator. It will generate many different topologies of netlists that implement the same logic function. To prevent that number of netlists from exploding, many of them can be embedded in a singular netlist using choice nodes or multi-input nets. Placement and routing can elaborate these candidates further such that one gets an understanding how these topologies will be physically implemented. The elaborated alternatives will be physically complete (placed and routed) and a much more reliable source for the analysis tools. Timing, power, and noise analysis will be our scoring engines. As soon as synthesis has generated the choices, the elaboration and scoring of each of them can be done in a very distributed fashion. A final stage scores all the partial alternatives and folds them together in one final result that meets the timing, power, and noise targets. Except for the final folding stage, all other steps can be implemented in a very parallel and distributed fashion. And even the final folding stage can be very parallel as long as the merging steps are very carefully executed.

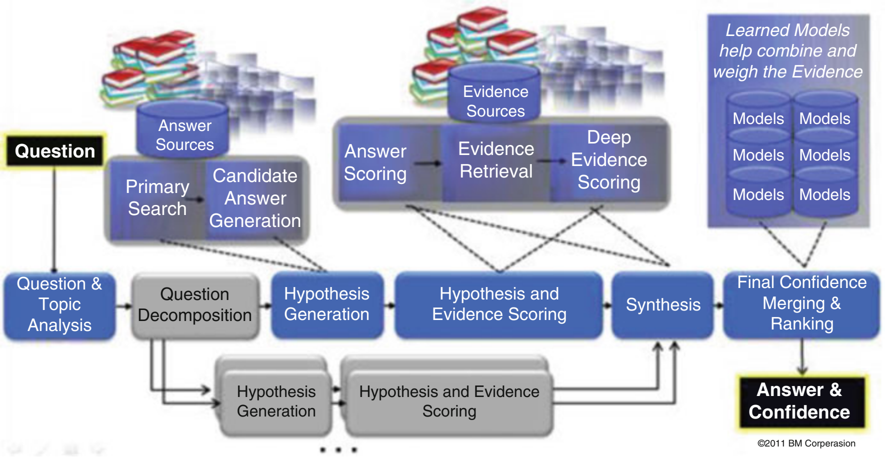


Fig. 1.9 DeepQA software architecture

Once we start to think about other optimization problems in EDA in this paradigm of "generating many alternatives and scoring them in a very distributed fashion" a larger part of the design space can be explored.

7 Summary

It is time for the design and EDA industry to embrace warehouse-scale computing. We need to look past cloud as an IT cost saving and look at warehouse-scale computing as an opportunity to bring a significant higher level of productivity to the design community. EDA3.0 tools need to be constructed such that the DATA is the central point in the EDA flow, not the optimization and analysis algorithms. These will become the services build around the data. Depending on their runtime they will need to be configured as online or offline services. Taking advantage of analytics technologies will drive to more intuitive and simplified user interfaces that provide better insight in the data. Using programming model Rest APIs will enforce standardized interface development. The massive compute power of a warehouse-scale computer will provide more scalable design solutions and better optimization. Analytics engines will allow for a much better insight in the derived design data than the numerous Perl and Python scripts currently provide.

EDA3.0 design flows will be significantly easier to manage and result in much more robust and predictable design flows. This will be necessary to take on the challenges to design large systems-on-chips built from future nano-scale devices.

References

1. L.A. Barroso, J. Clidaras, U. Hölzle, The datacenter as a computer: An introduction to the design of warehouse-scale machines. Synth. Lectures Comput. Architec. **8**(3), 1–154 (2013)
2. R. Brayton, J. Cong, Nsf workshop on EDA: Past, present, and future (part 2). IEEE Design Test Comput. **27**(3), 62–74 (2010)
3. J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
4. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks. ACM SIGOPS Oper. Syst. Rev. **41**(3), 59–72 (2007)
5. The Hadoop Project. [Online] http://hadoop.apache.org
6. R. Pike et al., Interpreting the data: Parallel analysis with Sawzall. Sci. Program. J. **13**, 227–298 (2005)
7. F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: A distributed storage system for structured data. ACM Transac. Comput. Syst. **26**(2), 4 (2008.) http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf
8. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: amazon's highly available key-value store. ACM SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007)
9. S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: Interactive analysis of web-scale datasets. Proc. VLDB Endowment **3**(1–2), 330–339 (2010)
10. J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, Spanner: Google's globally distributed database. ACM Transac. Comput. Syst. **31**(3), 8 (2012)
11. M. Burrows, W.A. Seattle, *The Chubby Lock Service for Loosely-Coupled Distributed Systems*, Proceedings of OSDI'06: Seventh Symposium on Operating System Design and Implementation, 2006
12. Openstack, [Online] www.openstack.org
13. IBM Spectrum LSF, [Online] IBM, http://www-03.ibm.com/systems/spectrum-computing/products/lsf/
14. OpenAccess, [Online] http://www.si2.org/openaccess/
15. Spark, [Online] https://en.wikipedia.org/wiki/Apache_Spark
16. Databricks, [Online] https://databricks.com
17. L. Rizzatti, *Digital Data Storage is Undergoing Mind-Boggling Growth*, [Online] http://www.eetimes.com/author.asp?section_id=36&doc_id=1330462
18. Google Maps facts, [Online] http://mashable.com/2012/08/22/google-maps-facts/
19. Data never sleeps, [Online] http://wersm.com/how-much-data-is-generated-every-minute-on-social-media/#!prettyPhoto
20. [Online] https://www.blazegraph.com/whitepapers/mapgraph_clusterBFS-ieee-bigdata-2014.pdf
21. [Online] http://spark-summit.org/2014/talk/quadratic-programing-solver-for-non-negative-matrix-factorization-with-spark
22. EDA forecast meeting, [Online] 14 Mar 2013, http://www.edac.org/events/2013-EDA-Consortium-Annual-CEO-Forecast-and-Industry-Vision/video
23. A. Kuehlmann, R. Camposano, J. Colgan, J. Chilton, S. George, R. Griffith, P. Leventis, D. Singh, *Does IC Design have a Future in the Clouds?*. Proceedings of the 47th Design Automation Conference, 2010 (pp. 412–414). ACM
24. [Online] https://www.cloudfoundry.org
25. [Online] https://console.ng.bluemix.net
26. [Online] Neo4j https://neo4j.com
27. L. Stok, M.A. Iyer, A.J. Sullivan, *Wavefront Technology Mapping*, Proceedings of the Conference on Design, Automation and Test in Europe, 1998 (p. 108), ACM

28. S. Chatterjee, A. Mishchenko, R.K. Brayton, X. Wang, T. Kam, Reducing structural bias in technology mapping. IEEE Transac. Comput. Aided Design Integr. Circuits Syst. **25**(12), 2894–2903 (2006)
29. [Online] http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2159
30. EDA file formats, [Online] En.wikipedia.org/wiki/CategoryEDA_file_formats

Chapter 2
Can Parallel Programming Revolutionize EDA Tools?

Yi-Shan Lu and Keshav Pingali

1 Introduction

I think this is the beginning of a beautiful friendship. Humphrey Bogart in *Casablanca*.

Until a decade ago, research in parallel programming was driven largely by the needs of computational science applications, which use techniques like the finite-difference and finite-element methods to find approximate solutions to partial differential equations. In finite differences, the key computational kernels are stencil computations on regular grids, and the solution of linear systems with structured sparsity such as banded systems. In finite-elements, the key computational kernel is the solution of sparse linear systems in which matrices have unstructured sparsity; these linear systems are usually solved using iterative methods like conjugate gradient in which the main computation is sparse matrix–vector multiplication.

Parallel programming research therefore focused largely on language support, compilation techniques, and runtime systems for matrix computations. Languages like High Performance FORTRAN (HPF) [21] and Coarray FORTRAN [32] were developed to make it easier to write matrix applications. Sophisticated compiler technology based on polyhedral algebra was invented to optimize loop nests that arise in matrix computations [6, 13, 14]. Runtime systems and communication libraries like OpenMP and MPI provided support for communication and synchronization patterns found in these applications.

While computational science applications and matrix computations continue to be important, our group at the University of Texas at Austin and several others across the world have shifted our focus to applications, such as the following ones, which compute on *unstructured graphs*.

Y.-S. Lu (✉) • K. Pingali
The University of Texas at Austin, Austin, TX, USA
e-mail: yishanlu@utexas.edu; pingali@cs.utexas.edu

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_2

- In social network analysis, the key data structures are extremely sparse graphs in which nodes represent entities and edges represent relationships between entities. Algorithms for breadth-first search, betweenness-centrality, page-rank, max-flow, etc. are used to extract network properties, to make friend recommendations, and to return results sorted by relevance for search queries [2, 20].
- Machine-learning algorithms like belief propagation and survey propagation are based on message-passing in a factor graph, which is a sparse bipartite graph [23].
- Data-mining algorithms like k-means and agglomerative clustering operate on dynamically changing sets and multisets [37].
- Traffic and battlefield simulations often use event-driven (discrete-event) simulation [27] in networks.
- Program analysis and instrumentation algorithms used within compilers are usually based on graphical representations of program properties, such as points-to graphs [3, 15, 25, 34].

Irregular graph applications such as these can have a lot of parallelism, but the patterns of parallelism in these programs are very different from the parallelism patterns one finds in computational science programs.

- Graphs in many of these applications are very dynamic data structures since their structure can be morphed by the addition and removal of nodes and edges during the computation. Matrices are not good abstractions for such graphs.
- Even if the graphs have fixed structure, many of the algorithms do not fit the matrix–vector/matrix–matrix multiplication computational patterns that are the norm in computational science. One example is delta-stepping, an efficient parallel single-source shortest-path (SSSP) algorithm [20, 26]. This algorithm maintains a work-list of nodes, partially sorted by their distance labels. Nodes enter and leave the work-list in a data-dependent, statically unpredictable order. This is a computational pattern one does not see in traditional computational science applications.
- Parallelism in irregular graph applications like delta-stepping is dependent not only on the input data but also on values computed at runtime. This parallelism pattern, which we call *amorphous* data-parallelism [33], requires parallelism to be found and exploited at runtime during the execution of the program. In contrast, the conventional data-parallelism in computational science kernels is independent of runtime values and can found by static analysis of the program.[1]

In spite of these difficulties, the parallel programming research community has made a lot of progress in the past 10 years in designing abstractions, programming models, compilers, and runtime systems for exploiting amorphous data-parallelism in graph applications. These advances have not yet had a substantial impact on EDA tools even though unstructured graphs underlie many EDA algorithms. The goal of

[1] Sparse direct methods are an exception, but even in these algorithms, a dependence graph, known as the elimination tree, is built before the algorithm is executed in parallel [5].

this paper is to summarize advances reported in previous papers [20, 33] and discuss their relevance to the EDA tools area, with the goal of promoting more interaction between the EDA tools and parallel programming communities.

The rest of this paper is organized as follows. Section 2 describes an abstraction for graph algorithms, called the operator formulation of algorithms. Section 3 discusses the patterns of parallelism in graph algorithms, and describes the Galois system, which exploits this parallelism while providing a sequential programming model implemented in C++. Section 4 summarizes the results of several case studies that use the Galois system, including scalability studies on large-scale shared-memory machines [18], and implementations of graph analytics algorithms [31], subgraph isomorphism algorithms, and FPGA maze routing [28]. We conclude in Sect. 5.

2 Abstractions for Graph Algorithms

Parallelism in matrix programs is usually described using program-centric concepts like parallel loops and parallel procedure calls. One lesson we have learned in the past 10 years is that parallelism in graph algorithms is better described using a *data-centric* abstraction called the *operator formulation of algorithms* in which data structures, rather than program constructs, play the central role [33]. To illustrate concepts, we use the single-source shortest-path (SSSP) problem. Given an undirected graph $G = (V, E, w)$ in which V is the set of nodes, E the set of edges, and w a map from edges to positive weights, the problem is to compute for each node the shortest distance from a source node s. There are many algorithms for solving this problem such as Dijkstra's algorithm, Bellman-Ford algorithm, delta-stepping and chaotic relaxation [20], but in the standard presentation, these algorithms appear to be unrelated to each other. In contrast, using the operator formulation elucidates their similarities and differences.

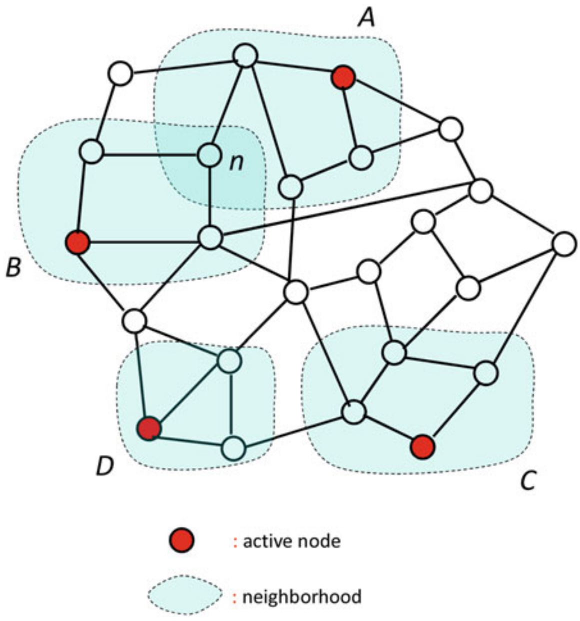
2.1 Operator Formulation

The operator formulation of an algorithm has a *local view* and a *global view*, shown pictorially in Fig. 2.1. This formulation of algorithms leads to a useful classification of algorithms, called TAO analysis, shown in Fig. 2.2.

2.1.1 Local View of Algorithms: Operators

The local view is described by an *operator*, which is a graph update rule applied to an *active node* in the graph (some algorithms have active *edges*, but to avoid verbosity, we refer only to active nodes in this paper). Each operator application,

Fig. 2.1 Operator formulation



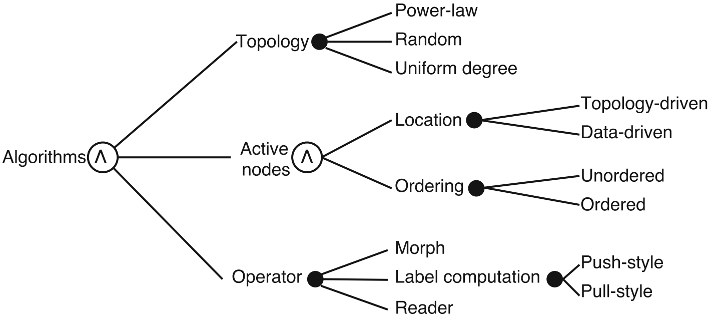


Fig. 2.2 TAO analysis of graph algorithms

called an *activity*, reads and writes a small region of the graph around the active node, called the *neighborhood* of that activity. In Dijkstra's SSSP algorithm, the operator, called the relaxation operator, uses the label of the active node to update the labels of its immediate neighbors. Figure 2.1 shows active nodes as filled dots, and neighborhoods as clouds surrounding active nodes, for a generic algorithm. An active node becomes inactive once the activity is completed.

In general, operators can modify the graph structure of the neighborhood by adding and removing nodes and edges (these are called *morph* operators). In most graph analytics applications, operators only update labels on nodes and edges, without changing the graph structure. These are called *label computation* operators; a *pull-style operator* reads the labels of nodes in its neighborhood and writes to the label of its active node, while a *push-style operator* reads the label of the active node and writes the labels of other nodes in its neighborhood. Dijkstra's algorithm uses a push-style operator. In algorithms that operate on several data structures, some data structures may be read-only in which case the operator is a *reader* for those data structures.

Neighborhoods can be distinct from the set of immediate neighbors of an active node, and in principle, can encompass the entire graph, although usually they are small regions of the graph surrounding the active node. Neighborhoods of different activities can overlap; in Fig. 2.1, node n is contained in the neighborhoods of both activities A and B. In a parallel implementation, the semantics of reads and writes to such overlapping regions, known as the *memory model*, must be specified carefully.

2.1.2 Global View of Algorithms: Location of Active Nodes and Ordering

The global view of a graph algorithm is captured by the location of active nodes and the order in which activities must appear to be performed.

Topology-driven algorithms make a number of sweeps over the graph until some convergence criterion is met; in each sweep, all nodes are active initially. The Bellman-Ford SSSP algorithm is an example. Data-driven algorithms, on the other hand, begin with an initial set of active nodes, and other nodes may become active on the fly when activities are executed. These algorithms do not make sweeps over the graph, and terminate when there is no more active nodes. Dijskstra's SSSP algorithm is a data-driven algorithm: initially, only the source node is active, and other nodes become active when their distance labels are lowered.

The second dimension of the global view of algorithms is *ordering*. In *unordered* algorithms, any order of processing active nodes is semantically correct; each sweep of the Bellman-Ford algorithm is an example. Some orders may be more efficient than others, so unordered algorithms sometimes assign *soft* priorities to activities, but these are only suggestions to the runtime system, and priority inversions are permitted in the execution. In contrast, *ordered* algorithms require that active nodes appear to have been processed in a specific order; Dijkstra's algorithm and algorithms for discrete-event simulation are examples. This order is specified by assigning priorities to active nodes, and the implementation is required to process active nodes so that they appear to have been scheduled for execution in *strict* priority order from earliest to latest.

2.2 Trade-offs Between Topology-Driven and Data-Driven Algorithms

Many graph problems, like SSSP, can be solved by both topology- and data-driven algorithms. However, one should be aware of the tradeoffs involved when choosing algorithms.

Topology-driven algorithms are easier to implement because iteration over active nodes can be implemented by traversing the nodes in the representation of the graph (usually arrays). However, there may be wasted work in each sweep because there may not be useful work done at many nodes.

In contrast, data-driven algorithms can be work-efficient since activities are performed where there is useful work to be done. For many problems, data-driven algorithms are asymptotically faster than topology-driven algorithms. For instance, the complexity of the Bellman-Ford algorithm is $O(|E||V|)$, whereas Dijkstra's algorithm is $O(|E|\log(|V|))$.

On the other hand, data-driven algorithms can be complicated to implement because they need a work-set to track active nodes. Sequential implementations use lists and priority queues for unordered and ordered algorithms, respectively. Concurrent work-lists for graph algorithms are difficult to implement efficiently: since the amount of work in each activity is usually fairly small, adding and removing active nodes from the work-list can become a bottleneck unless the work-list is designed very carefully. Nguyen et al. [19, 31] describe a scalable work-set called *obim* that supports soft priorities.

The best choice of algorithm for a given problem can also depend on the topology of the graph, as we show in Sect. 4.2 [9, 29]. In many social networks such as the web graph or the Facebook friends graph, the degree distribution of nodes roughly follows a power law, so these are often referred to as *power-law* graphs. In contrast, road networks and 2D/3D grids/meshes are known as *uniform-degree* graphs because most nodes have roughly the same degree. Graphs for VLSI circuits fall in this category. *Random* graphs, which are created by connecting randomly chosen pairs of nodes, constitute another category of graphs. Different graph classes have very different properties: for example, the diameter of a randomly generated power-law graph grows only as the logarithm of the number of nodes in the graph but for uniform-degree graphs, the diameter can grow linearly with the number of nodes [20].

Figure 2.2 summarizes this discussion. We call it TAO analysis for its three main dimensions: Topology of the input graph, Activity location and ordering, and Operator. Note that TAO analysis does not distinguish between sequential and parallel algorithms.

3 Exploiting Parallelism in Graph Algorithms

Parallelism can be exploited by processing active nodes in parallel, subject to neighborhood and ordering constraints. Since neighborhoods can overlap, the memory model, which defines the semantics of reads and writes in overlapped regions, may prevent some activities with overlapping neighborhoods from executing in parallel. In addition, ordering constraints between activities must be enforced. We call this pattern of parallelism *amorphous* data-parallelism [33]; it is a generalization of data-parallelism in which (1) there may be neighborhood and ordering constraints that

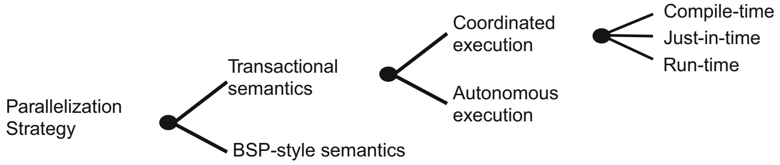


Fig. 2.3 Parallelization strategies

prevent all activities from executing in parallel, and (2) the execution of an activity may create new activities.

Figure 2.3 summarizes the important choices in implementing parallel graph programs. There are two popular memory models: BSP-style (Bulk-Synchronous Parallel) semantics [39] and transactional semantics.

3.1 BSP-Style Semantics

The program is executed in rounds (also known as super-steps), with barrier synchronization between rounds. Writes to the graph are considered to be communication from a round to the following round, so they are applied to the graph only at the beginning of the following round. Multiple updates to a label are resolved as in PRAM models such as by using a reduction operation to combine the updates into a single update [11].

BSP-style parallelization may work well for graph applications in which the number of activities in each round is large enough to keep the processors of the parallel machine busy. One example is breadth-first search (BFS) on a power-law graph. Each round handles nodes at a single BFS level and computes labels for nodes at the next BFS level. Since the average diameter of power-law graphs is small, there will be a lot of parallel activities in most rounds. On the other hand, BSP-style parallelization may not perform well for graphs that have high average diameter, such as road networks or VLSI circuits, as we show experimentally in Sect. 4.2. This is because the number of super-steps required to execute the algorithm may be large, and the number of activities in each super-step may be small.

3.2 Transactional Semantics

In this model, parallel execution of activities is required to produce the same answer as executing activities one at a time in some order that respects priorities. Intuitively, this means that activities should not “see” concurrently executing activities, and the updates made by an activity should become visible to other activities only after that activity completes execution. Formally, these two properties of transactional execution are known as isolation and atomicity.

Transactional semantics are implemented by preventing activities from executing in parallel if they *conflict*. For unordered algorithms, a conservative definition is that activities conflict if their neighborhoods overlap. In Fig. 2.1, activities *A* and *B* conflict because node *n* is in both their neighborhoods. Activities *C* and *D* do not conflict, and they can be executed in parallel with either *A* or *B*. Exploiting properties of the operator such as commutativity can lead to more relaxed definitions of conflicts, enhancing parallelism [16]. Given a definition of conflicts, the implementation needs to ensure that conflicting activities do not update the graph in parallel. This can be accomplished using *autonomous scheduling* or *coordinated scheduling*.

In autonomous scheduling, activities are executed speculatively. If a conflict is detected with other concurrently executing activities, some activities are aborted, enabling others to make progress; otherwise, the activity commits, and its updates become visible to other activities. Autonomous scheduling is good for exploiting parallelism but for some unordered algorithms, the output of the program can depend on the precise order in which activities are executed so the output may be non-deterministic in the sense that different runs of the program for the same input may produce different outputs. Delaunay mesh refinement and maze routing, discussed in Sect. 4.4, are examples. It is important to notice that this non-determinism, known as *don't-care non-determinism*, arises from under-specification of the order in which activities must be processed, and not from race conditions in updating shared state: even in a sequential implementation, the output of the program can depend on the order in which the work-list of active nodes is processed.

Coordinated scheduling strategies ensure that conflicting activities do not execute simultaneously [33]. For some algorithms, such as those that can be expressed using generalized sparse matrix–vector product, static analysis of the operator shows that active nodes can be executed in parallel without any conflict-checking. This is called *static parallelization*, and it is similar to auto-parallelization of dense array programs. *Just-in-time parallelization* preprocesses the input graph to find conflict-free schedules (e.g., by graph coloring); it can be used for topology-driven algorithms like Bellman-Ford in which neighborhoods are independent of data values. *Runtime parallelization* is general: the algorithm is executed in a series of rounds, and in each round, a set of active nodes is chosen, their neighborhoods are computed, and a set of non-conflicting activities are selected and executed. This approach can be used for deterministic execution of unordered algorithms [31].

3.3 The Galois System

The Galois system is an implementation of these data-centric abstractions.[2] Application programmers write programs in sequential C++, using certain programming

[2] A more detailed description of the implementation of the Galois system can be found in our previous papers such as [31].

```
 1  #include "Galois/Galois.h"
 2
 3  struct Data {int dist;};
 4  typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;
 5  typedef Graph::GraphNode Node;
 6
 7  struct BFS {
 8    Graph& g;
 9    BFS(Graph& g) : g(g) {}
10    void operator()(Node n, Galois::UserContext<Node>& ctx) {
11      int newDist = g.getData(n).dist + 1;
12      for(auto e : g.edges(n)) {
13        Node dst = g.getEdgeDst(e);
14        int& dstDist = g.getData(dst).dist;
15        if(dstDist > newDist) {
16          dstDist = newDist;
17          ctx.push(dst);
18        }
19      }
20    }
21  };
22
23  int main(int argc, char** argv) {
24    Graph g;
25    Galois::readGraph(g, argv[1]);
26    Galois::do_all_local(g, [&g] (Node n) {g.getData(n).dist = DIST_INFINITY;});
27    int start = atoi(argv[2]);
28    Node src = *(std::advance(g.begin(),start));
29    g.getData(src).dist = 0;
30    Galois::for_each(src, BFS{g});
31    return 0;
32  }
```

Fig. 2.4 Push-style BFS in Galois

patterns, described below, to highlight opportunities for exploiting amorphous data-parallelism.

Key features of the system are described below, using the code for push-style BFS shown in Fig. 2.4. This code begins by reading a graph from a file (line 25) and constructing a compressed-sparse-row (CSR) representation in memory (line 4). Line 26 initializes the `dist` fields of all nodes to ∞, and lines 27–29 read in the ID of the source node and initialize its `dist` field to 0.

- Application programmers specify parallelism *implicitly* by using Galois set iterators [33] which iterate over a work-list of active nodes. For data-driven algorithms, the work-list is initialized with a set of active nodes before the iterator begins execution. The execution of a iteration can create new active nodes, and these are added to the work-list when that iteration completes execution. Topology-driven algorithms are specified by iteration over graph nodes, and the iterator is embedded in an ordinary (sequential) loop, which iterates until the convergence criterion is met. In Fig. 2.4, data-driven execution is specified by line 30, which uses a Galois set iterator to iterate over a work-list initialized to contain the source node `src`.
- The body of the iterator is the implementation of the operator, and it is an imperative action that reads and writes global data structures. In Fig. 2.4, the operator is specified in lines 10–20. This operator iterates over all the neighbors

of the active node, updating their distance labels as needed. Iterations are required to be *cautious*: an iteration must read *all* elements in its neighborhood before it writes to *any* of them [33]. In our experience, this is not a significant restriction since the natural way of writing graph analytics applications results in cautious iterations.

- For unordered algorithms, the relative order in which iterations are executed is left unspecified in the application code. An optional application-specific priority order for iterations can be specified with the iterator [30], and the implementation tries to respect this order when it schedules iterations.
- The system exploits parallelism by executing iterations in parallel. To ensure serializability of iterations, programmers must use a library of built-in concurrent data structures for graphs, work-lists, etc. These library routines expose a standard API to programmers, and they implement lightweight synchronization to ensure serializability of iterations, as explained below.

Inside the data structure library, the implementation of a data structure operation such as reading a graph node or adding an edge between two nodes acquires logical locks on nodes and edges before performing the operation. If the lock is already owned by another iteration, the iteration that invoked the operation releases all of its acquired locks and is rolled back; it is retried again later. Intuitively, the cautiousness of iterations reduces the synchronization problem to the dining philosopher's problem [4], obviating the need for more complex solutions like transactional memory. The system also supports BSP-style execution of activities, and this can be specified by the user using a directive for the iterator. This is useful for deterministic execution of unordered algorithms.

4 Using Galois: Case Studies

This section discusses a number of case studies in which the Galois system is used to parallelize algorithms from several domains. Section 4.1 shows the scalability of Galois programs for HPC and graph analytics algorithms on a large-scale NUMA shared-memory machine. Section 4.2 compares Galois program performance with the performance of programs in Ligra [36] and PowerGraph [7], two popular shared-memory graph analytics systems. We show that for road networks, which are high-diameter graphs like circuit graphs, Galois programs run orders of magnitude faster than Ligra and PowerGraph programs. Section 4.3 describes implementations of subgraph matching algorithms in Galois. Finally, Sect. 4.4 describes how the Galois system was used by Moctar and Brisk to perform parallel FPGA maze routing [28].

4.1 Case Study: Large-Scale Shared-Memory Machines

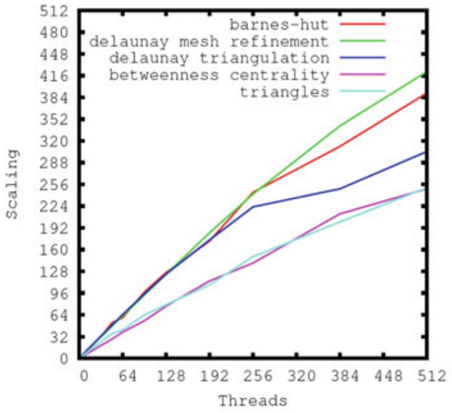
In this section, we summarize the results of a study by Lenharth and Pingali on the performance of Galois programs on a large-scale NUMA shared-memory machine [18]. The machine used in this study is the Pittsburgh Supercomputing Center's Blacklight system, which is an SGI UltraViolet NUMA system with 4096 cores and 32 TiB of ram (our machine allocation was limited to 512 cores). Each NUMA node contains 16 cores running at 2.27 GHz on two packages and 128 GiB of memory. We compile using g++ 4.7 at -O3. The benchmarks used are Barnes-Hut (`bh`), an n-body simulation code; Delaunay mesh generator (`dt`), a guaranteed-quality 2D triangular mesh generator; Delaunay mesh refinement (`dmr`), a mesh refinement algorithm for 2D meshes; betweenness centrality (`bc`), a centrality computation in networks; and triangle counting (`tri`), which counts the number of triangles in a graph. Table 2.1 summarizes the inputs and configurations used. Although these results are on the SGI machine, similar results are seen on smaller scale NUMA systems.

Figure 2.5 shows that `dmr` and `bh` achieve self-relative, strong scaling of 422× and 390×, respectively, at 512 threads. This equates to an 82% and 75% parallel efficiency for programs written in a sequential programming style. Delaunay triangulation scales only to 304× at 512 threads, due in part to memory contention when inserting into the lookup-acceleration tree. Betweenness centrality requires reading the entire graph in each iteration. Although the graph size is small enough to fit in the L3 cache, the temporary data necessary for an "outer-loop" parallel `bc` calculation is proportional to the size of the graph, so in actual parallel execution, the graph could not remain in cache. Adding NUMA nodes increases the number of cores but hurts the average latency of memory accesses for all threads, and this causes `bc` to scale at only about 50% efficiency. Triangle finding scales at about 50% efficiency.

Scaling numbers can be deceptive because they do not take into account the effect of single-thread overheads. Therefore, we also compare the single-threaded performance of the Galois codes to third-party serial implementations of these algorithms. Our goal is not necessarily to have the best performing serial implementation, especially since some of the implementations use hand-crafted,

Table 2.1 Inputs used in evaluation on SGI Ultraviolet

App	Input and configuration
`bh`	10 million bodies generated using a plummer model, tolerance = 0.05, timestep = 0.5, eps = 0.05
`dmr`	20 million triangles in a square, 50% bad
`dt`	10 million points randomly distributed in a square
`bc`	Random graph with average degree 4 and 2^{18} nodes
`tri`	Random planar graph with average degree 4 and 2^{28} nodes



App	Implementation	Threads	Time (s)
dmr	triangle	1	96
	Galois	1	155.7
	Galois	512	0.37
dt	triangle	1	1185
	Galois	1	56.6
	Galois	512	0.18
bh	splash2	1	>6000
	Galois	1	1386
	Galois	512	3.55
bc	HPCS SSCA	1	6720
	Galois	1	5394
	Galois	512	21.6
tri	graphlab	2	531
	Galois	1	7.03
	Galois	512	0.028

Fig. 2.5 Performance of Galois programs: SGI Ultraviolet

problem-specific data-structures, but to show that we are within an acceptable margin of custom implementations even on one thread while using the generic data-structures provided by our runtime.

Figure 2.5 shows that we compare favorably with third party implementations for all our benchmarks. For Delaunay triangulation and Delaunay mesh refinement, we compare to Triangle [35]. Our implementation of refinement on one thread is slower than Triangle, but triangulation is much faster (due to a more efficient algorithm). For Barnes-Hut, we compare to SPLASH-2 [40]. Although the SPLASH implementation is an ancestor of our implementation, ours is slightly faster. For betweenness centrality, we compare to the Scalable Synthetic Compact Applications benchmark suite [1] and we are 20% faster. Finally, compared to the nearly identical implementation in GraphLab [22], our implementation of triangles on 1 thread is 75× faster than Graphlab on 2 threads (the Graphlab code did not terminate when it was run on 1 thread).

4.2 Case Study: Graph Analytics

Parallel graph analytics has become a popular area of research in the past few years. In these applications, labels on nodes are repeatedly updated until some convergence criterion is reached, but the graph structure is not modified (in the TAO classification described in Fig. 2.2, these algorithms use label computation operators). Nguyen et al. [31] compared the performance of graph analytics applications written in Galois with the performance of the same applications written in two other frameworks, PowerGraph [7] and Ligra [36]. We summarize their study in this section.

PowerGraph [7] is a programming model for vertex programs, an abstraction in which the neighborhood of an active node is limited to itself and the set of its immediate neighbors [24]. It supports shared-memory parallelism in a single machine as well as distributed-memory execution on clusters. *Ligra* [36] is a shared-memory programming model for vertex programs. Ligra is capable of switching between pull- and push-style operators during execution time to improve cache utilization. Both PowerGraph and Ligra support only bulk synchronous parallelism (BSP).

Unlike most graph analytics studies, the study of Nguyen et al. used both power-law graphs (twitter-50 with 51 million nodes and 2 billion edges) and road networks (U.S. road network with 24 million nodes and 58 million edges). *Graphs of importance in the EDA tools area, such as circuit graphs, are likely to be high-diameter graphs similar to road networks, so this study sheds some light on what kinds of graph processing systems are likely to be useful for parallel EDA tools.*

Nguyen et al. used the following applications in their study:

Single-source shortest-paths (`SSSP`) is the problem used in Sect. 2 to illustrate the operator formulation of algorithms.

Breadth-first search (`BFS`) is a special case of `SSSP` in which all edge weights are one.

Approximate diameter (`DIA`) computes an approximation to the graph diameter, which is the maximum length of the shortest distance between all pairs of nodes.

Connected components (`CC`) divides the nodes of an undirected graph into equivalence classes by reachability.

Pagerank (`PR`) computes a relative importance score for each node in a graph.

Figure 2.6 compares the performance of the three graph analytics frameworks with different pairs of applications and input graphs. The experiments were run on a machine with 40-core Intel E7-4860 and 128 GB of memory. Notice that the y-axis is a log-scale.

Although the road network is roughly 40 times smaller than the Twitter graph, Ligra and PowerGraph take far more time for `BFS` and `SSSP` on the road network than on the Twitter graph. The U.S. road network has a large diameter and a uniform, low degree distribution, so BSP-style implementation of algorithms requires a large number of low-parallelism rounds. Galois avoids this problem by providing asynchronous scheduling of activities, and is orders of magnitude faster than Ligra and PowerGraph.

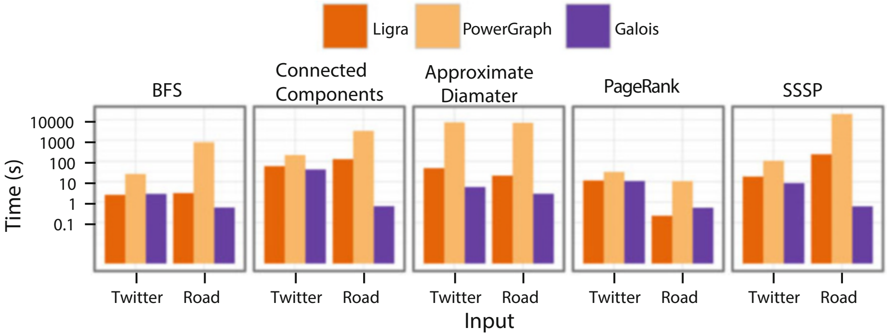


Fig. 2.6 Comparison of graph analytics frameworks [31]

The performance of CC highlights the value of Galois's support of operators with arbitrary neighborhoods. Since Ligra and PowerGraph support only vertex programs, CC on these systems requires a label-propagation algorithm: all nodes are given distinct component IDs, and the nodes with smallest IDs propagate their IDs to all nodes in their components. On the other hand, the Galois program uses a parallel union-find data structure. The union-find data structure is updated by pointer jumping, which is similar to the find operation in disjoint-set union-find [11, 12]. However, pointer jumping cannot be expressed as a vertex program, so the program cannot be written using Ligra and PowerGraph.

In summary, runtimes vary widely for different programming models even for the same problem. Pagerank demonstrates the least variation since all three frameworks use a topology-driven algorithm. For other problems, systems like Ligra and PowerGraph that are based on BSP-style execution of vertex programs can be several orders of magnitude slower than Galois, especially for road networks.

4.3 Case Study: Subgraph Isomorphism

Subgraph isomorphism is an important kernel for many applications; for example, it can sometimes be used to locate and then replace a suboptimal circuit with a functionally equivalent circuit with better delay, area, or power consumption. Formally, subgraph isomorphism is defined as follows [38]: given a query graph $G^q = (V^q, E^q)$ and a data graph $G^d = (V^d, E^d)$, a subgraph isomorphism exists between G^q and G^d if and only if there exists a function $I : V^q \rightarrow V^d$ such that

1. $i \neq j \implies I(v_i^q) \neq I(v_j^q)$, and
2. $(v_i^q, v_j^q) \in E^q \implies (I(v_i^q), I(v_j^q)) \in E^d$.

Intuitively, the function I, which maps query graph nodes to data graph nodes, is (1) injective, and (2) if two query nodes are connected by an edge, their images in the data graph are also connected by an edge. This definition assumes unlabeled graphs; it can be extended to deal with graphs with labeled nodes and edges. Subgraph isomorphism can be solved trivially by a generate-and-test approach in which all possible injective functions mapping query graph nodes to data graph nodes are generated, and each one is tested to see if it satisfies condition (2). To make this approach tractable, we must avoid generating, whenever possible, mappings that fail the test. There are many heuristics for this, and they can be described abstractly by the template shown in Algorithm 1 [17].

Procedure GenericGraphQuery preprocesses the graph by determining for each query node, an initial set of candidate data nodes that it can be mapped to, based on properties such as node labels, degrees, etc. If any query node has an empty set of candidate data nodes, then no subgraph isomorphism can be found. Otherwise, we call procedure SubgraphSearch to compute all subgraph isomorphisms, starting from an empty matching M.

Algorithm 1 Generic graph query algorithm for finding subgraph isomorphism

```
Input: $G^q$, query graph; $G^d$, data graph.
Output: All $I : V^q \rightarrow V^d$ satisfying subgraph isomorphism.
procedure GenericGraphQuery($G^q, G^d$)
    for all $v_i^q \in G^q$ do
        $C_{v_i^q} \leftarrow$ FilterCandidates($v_i^q, G^q, G^d, \ldots$)
        if $C_{v_i^q} = \emptyset$ then
            return
        end if
    end for
    $M \leftarrow \emptyset$
    SubgraphSearch($G^q, G^d, M, \ldots$)
end procedure
procedure SubgraphSearch($G^q, G^d, M, \ldots$)
    if $|M| = |V^q|$ then
        report $V^q \rightarrow M$ in matching order
        return
    end if
    $v_j^q \leftarrow$ NextQueryNode($G^q, G^d, M \ldots$)
    $R \leftarrow$ RefineCandidates($v_j^q, G^q, C_{v_j^q}, G^d, M \ldots$)
    for all $v_k^d \in R$ do
        if IsJoinable($v_j^q, G^q, v_k^d, G^d, M \ldots$) then
            update $M$
            SubgraphSearch($G^q, G^d, M \ldots$)
            restore $M$
        end if
    end for
end procedure
```

In Procedure `SubgraphSearch`, the call to `NextQueryNode` heuristically chooses v_j^q, the next query node to be matched. Next, procedure `RefineCandidates` refines R, the set of candidate data nodes for v_j^q, based on current matching. Finally, for every data node v_k^d in R, procedure `IsJoinable` checks if v_k^d can be joined to current matching M and can satisfy all edge constraints. If so, we add v_k^d to M, recurse on `SubgraphSearch`, and remove v_k^d from M when the recursive call terminates.

Algorithms for subgraph isomorphism differ in the design of procedures `FilterCandidates`, `NextQueryNode`, and `RefineCandidates`.

- The *Ullmann* algorithm [17] constructs initial sets of candidate data nodes based on node labels, follows input order for deciding next query node to be matched, and does no refinement for sets of candidate data nodes in `SubgraphSearch`.

- The *VF2* algorithm [17] starts search from the first query node. It considers only unmatched immediate neighbors of matched query/data nodes when choosing next query node and refining sets of candidate data nodes. Also, a candidate data node is discarded by `RefineCandidates` if its number of unmatched neighbors is fewer than that of its query counterpart.

To parallelize these two algorithms, we view the search process as traversing a search tree whose nodes are query nodes being matched and branches (edges) are data nodes matched to the corresponding query nodes. Different subtrees share no information with each other, and the search process does not modify the query and data graphs. Therefore, we parallelize Ullmann and VF2 algorithms along the first level of the underlying search tree. Each task is a pair consisting of the first query node and one of its candidate data nodes. Threads claim tasks from a work queue, which is populated with all tasks initially. Using this parallelization strategy, Ullmann and VF2 algorithms are unordered, data-driven algorithms using reader operators.

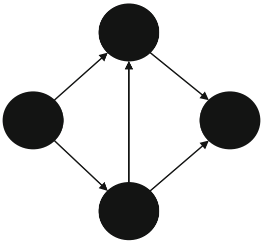
The Ullmann and VF2 algorithms are implemented using Galois 2.3.0 beta with Boost library ver. 1.58.0. Experiments are run on a Linux server with 32 cores of Intel Xeon E7520 running at 1.87 GHz and 64GB RAM. We use two data graphs to run our experiments. RoadNY is the road network of New York City, and THEIA is a graph extracted from operating system activity logs. Table 2.2 summarizes the statistics for the data graphs. Figure 2.7 shows the query graph used in our experiments. All data graphs and the query graph are directed and unlabeled.

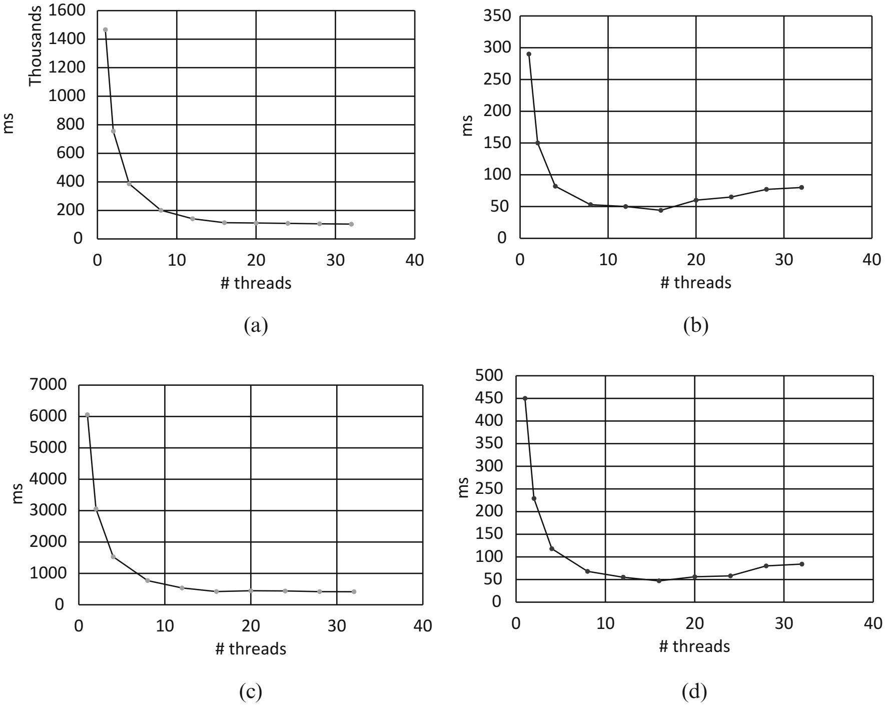
Figure 2.8 shows performance numbers. For roadNY, the Ullmann algorithm scales well but the VF2 algorithm outperforms it significantly. Since the sizes of frontiers, i.e. the number of unmatched immediate neighbors of matched nodes, grow slowly in road networks, filtering by frontiers works well. The scaling for VF2 is not as good as the scaling for Ullmann since the graphs are small. For THEIA, VF2 still outperforms Ullmann but not as much as when the data graph is roadNY.

Table 2.2 Data graphs for subgraph isomorphism

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Est. diameter
roadNY	264,346	730,100	720
THEIA	7,478	17,961	2

Fig. 2.7 The query graph for subgraph isomorphism





(a)

(b)

(c)

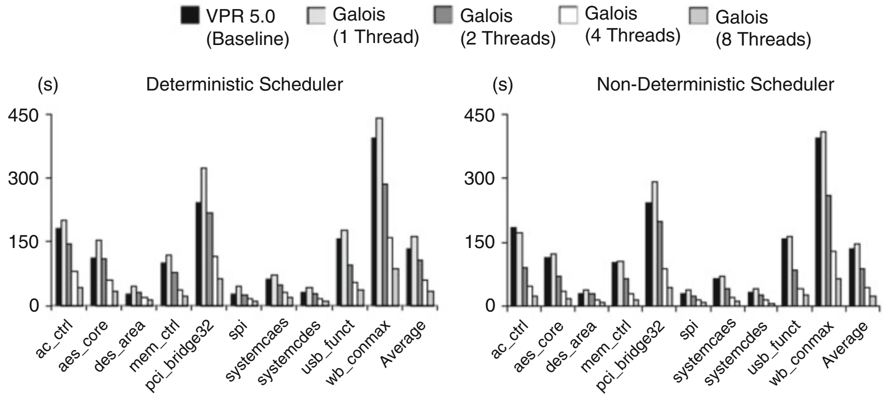
(d)

Fig. 2.8 Performance for finding 100 instances of the query graph in data graphs; (**a**) in roadNY by ullmann, (**b**) in roadNY by vf2, (**c**) in THEIA by ullmann, (**d**) in THEIA by vf2

Since THEIA is more similar to a power-law graph, its frontier grows rapidly, a fact which renders VF2's filtering by frontiers less effective in reducing the number of possible candidates to match.

4.4 Case Study: Maze Routing in FPGAs

Moctar and Brisk have used the Galois system to parallelize PathFinder, the routing algorithm used in many commercial FPGA tool chains [28]. Previous studies [8] have shown that roughly 70% of the execution time of PathFinder is spent in *maze expansion*, which performs an A* search of a routing resource graph (RRG) to route signals through the chip. This search procedure can be parallelized, but if different nets end up sharing routing resources, the resulting solution is illegal. When this happens, PathFinder needs to back up and reroute some nets to restore legality. PathFinder fails if a legal solution is not discovered within a user-specified number of iterations.



Averages				
	VPR 5.0	134.6 seconds	VPR 5.0	134.6 seconds
	Galois (1 Thread)	162.4 seconds	Galois (1 Thread)	145.3 seconds
	Galois (2 Threads)	106.6 seconds	Galois (2 Threads)	88.8 seconds
	Galois (4 Threads)	59.2 seconds	Galois (4 Threads)	43.0 seconds
	Galois (8 Threads)	33.7 seconds	Galois (8 Threads)	22.6 seconds

Fig. 2.9 Performance of Maze Router in Galois [28]

This computation pattern maps well to the asynchronous optimistic execution model described in Sect. 3. Moctar and Brisk evaluated their implementation on ten of the largest IWLS benchmarks [10], using an 8-core Intel Xeon platform. As a baseline, circuits were generated for these benchmarks using the publicly available Versatile Placement and Routing (VPR) system. A rough idea of the size of these circuits can be obtained from the fact that one of the largest circuits, generated for the `wb_conmax` input, had 10,430 nets and 6,297 logic blocks. Key findings from the study include the following.

- The Galois implementation was able to successfully route all of the nets in all benchmarks.
- The average speed-up over the serial VPR implementation was roughly 3 for 4 threads, and 5.5 for 8 threads, as shown in Fig. 2.9.
- The implementation was roughly three times faster than the best previous implementation of parallel multi-threaded FPGA routing.
- Because of the don't-care non-determinism in autonomous scheduling, different runs of the Galois program can produce different routing solutions, but the variation in critical path delay was small.

To eliminate don't-care non-determinism, the study also used the deterministic, round-based execution of Galois programs described in Sect. 3. This reduced speed-up to 4 on 8 threads, but ensured that each run of the program, even on different numbers of cores, returned the same routing solution.

Moctar and Brisk summarized their results as follows: "we *strongly believe* that Galois' approach is the *right* solution for parallel CAD, due to the widespread use of graph-based data structures (e.g. netlists) that exhibit irregular parallelism" [28].

5 Conclusions

In the past decade, there has been a lot of progress in understanding the structure of parallelism in graph computations, and there is now a rich literature on programming notations, compilers, and runtime systems for large-scale graph computations. Meanwhile, circuit designs have become sufficiently complex that the EDA tools community has begun to look at parallel computation as a way of speeding up the design process. Therefore the time has come for closer interaction between the EDA tools and parallel programming communities. We hope this paper catalyzes this interaction.

Acknowledgements This research was supported by NSF grants 1218568, 1337281, 1406355, and 1618425, and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563.

References

1. D.A. Bader, K. Madduri, Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors, in *High Performance Computing, HiPC'05* (2005)
2. U. Brandes, T. Erlebach (eds.), *Network Analysis: Methodological Foundations* (Springer, Heidelberg, 2005)
3. G. Bronevetsky, D. Marques, K. Pingali, P. Stodghill, C3: a system for automating application-level checkpointing of MPI programs. *Languages and Compilers for Parallel Computing* (Springer, New York, 2004), pp. 357–373
4. K.M. Chandy, J. Misra, The drinking philosophers problem. ACM Trans. Program. Lang. Syst. **6**(4), 632–646 (1984)
5. I.S. Duff, A.M. Erisman, J.K. Reid, *Direct Methods for Sparse Matrices* (Oxford University Press, New York, 1986). ISBN 0-198-53408-6
6. P. Feautrier, Some efficient solutions to the affine scheduling problem: one dimensional time. Int. J. Parallel Prog. **21**(5), 313–347 (1992)
7. J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, Berkeley, CA, 2012, pp. 17–30. USENIX Association. ISBN 978-1-931971-96-6. http://dl.acm.org/citation.cfm?id=2387880.2387883
8. M. Gort, J. Anderson, Deterministic multicore parallel routing for FPGAs, in *International Conference on Field Programmable Technology (ICFPT'10)* (2010)
9. M.A. Hassaan, M. Burtscher, K. Pingali, Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms, in *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming, PPoPP '11* (ACM, New York, 2011), pp. 3–12. ISBN 978-1-4503-0119-0. doi:http://doi.acm.org/10.1145/1941553.1941557. http://iss.ices.utexas.edu/Publications/Papers/ppopp016s-hassaan.pdf
10. IWLS, IWLS 2005 benchmarks. http://iwls.org/iwls2005/benchmarks.html (2005)
11. J. JaJa, *An Introduction to Parallel Algorithms* (Addison-Wesley, Boston, 1992)
12. R.M. Karp, V. Ramachandran, A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD-88-408, EECS Department, University of California, Berkeley (1988). http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/5865.html
13. K. Kennedy, J. Allen (eds.) *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach* (Morgan Kaufmann, San Francisco, 2001)

14. I. Kodukula, N. Ahmed, K. Pingali, Data-centric multi-level blocking, in *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (ACM, New York, 1997), pp. 346–357. ISBN 0-89791-907-6. doi:http://doi.acm.org/10.1145/258915.258946. http://iss.ices.utexas.edu/Publications/Papers/PLDI1997.pdf
15. V. Kotlyar, K. Pingali, P. Stodghill, A relational approach to the compilation of sparse matrix programs, in *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing* (Springer, London, 1997), pp. 318–327. ISBN 3-540-63440-1. http://iss.ices.utexas.edu/Publications/Papers/EUROPAR1997.pdf
16. M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, K. Pingali, Exploiting the commutativity lattice, in *PLDI* (2011)
17. J. Lee, W.-S. Han, R. Kasperovics, J.-H. Lee, An in-depth comparison of subgraph isomorphism algorithms in graph databases. Proc. VLDB Endow. **6**(2), 133–144 (2012). ISSN 2150-8097. doi:10.14778/2535568.2448946. http://dx.doi.org/10.14778/2535568.2448946
18. A. Lenharth, K. Pingali, Scaling runtimes for irregular algorithms to large-scale NUMA systems. Computer **48**(8), 35–44 (2015)
19. A. Lenharth, D. Nguyen, K. Pingali, Priority queues are not good concurrent priority schedulers, in *European Conference on Parallel Processing* (Springer, Berlin/Heidelberg, 2015), pp. 209–221
20. A. Lenharth, D. Nguyen, K. Pingali, Parallel graph analytics. Commun. ACM **59**(5), 78–87 (2016). ISSN 0001-0782. doi:10.1145/2901919. http://doi.acm.org/10.1145/2901919
21. D.B. Loveman, High performance fortran. IEEE Parallel Distrib. Technol. Syst. Appl. **1**(1), 25–42 (1993)
22. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, Graphlab: a new parallel framework for machine learning, in *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010
23. D. Mackay, *Information Theory, Inference and Learning Algorithms* (Cambridge University Press, Cambridge, 2003)
24. G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing - "abstract", in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC '09* (ACM, New York, 2009), pp. 6–6 ISBN 978-1-60558-396-9. doi:10.1145/1582716.1582723. http://doi.acm.org/10.1145/1582716.1582723
25. M. Mendez-Lojo, A. Mathew, K. Pingali, Parallel inclusion-based points-to analysis, in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, October 2010. http://iss.ices.utexas.edu/Publications/Papers/oopsla10-mendezlojo.pdf
26. U. Meyer, P. Sanders, Delta-stepping: a parallel single source shortest path algorithm, in *Proceedings of the 6th Annual European Symposium on Algorithms, ESA '98* (Springer, London, 1998), pp. 393–404. ISBN 3-540-64848-8. http://dl.acm.org/citation.cfm?id=647908.740136
27. J. Misra, Distributed discrete-event simulation. ACM Comput. Surv. **18**(1), 39–65 (1986), ISSN 0360-0300. doi:http://doi.acm.org/10.1145/6462.6485
28. Y.O.M. Moctar, P. Brisk, Parallel FPGA routing based on the operator formulation, in *Proceedings of the 51st Annual Design Automation Conference, DAC '14* (2014).
29. R. Nasre, M. Burtscher, K. Pingali, Data-driven versus topology-driven irregular computations on GPUs, in *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium, IPDPS '13* (Springer, London, 2013)
30. D. Nguyen, K. Pingali, Synthesizing concurrent schedulers for irregular algorithms, in *Proceedings of International Conference Architectural Support for Programming Languages and Operating Systems, ASPLOS '11*, pp. 333–344 (2011). ISBN 978-1-4503-0266-1. doi:10.1145/1950365.1950404. http://doi.acm.org/10.1145/1950365.1950404

31. D. Nguyen, A. Lenharth, K. Pingali, A lightweight infrastructure for graph analytics, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13* (ACM, New York, 2013), pp. 456–471. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522739. http://doi.acm.org/10.1145/2517349.2522739
32. R.W. Numrich, J. Reid, Co-array fortran for parallel programming, in *ACM Sigplan Fortran Forum*, vol. 17 (ACM, New York, 1998), pp. 1–31
33. K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M.A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, X. Sui, The TAO of parallelism in algorithms, in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pp. 12–25 (2011). ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993501. http://doi.acm.org/10.1145/1993498.1993501
34. D. Prountzos, R. Manevich, K. Pingali, K.S. McKinley, A shape analysis for optimizing parallel graph programs, in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11* (ACM, New York, 2011), pp. 159–172. ISBN 978-1-4503-0490-0. doi:http://doi.acm.org/10.1145/1926385.1926405. http://www.cs.utexas.edu/users/dprountz/popl2011.pdf
35. J.R. Shewchuk, Triangle: engineering a 2D quality mesh generator and delaunay triangulator, in *Applied Computational Geometry: Towards Geometric Engineering*, ed. by M.C. Lin, D. Manocha. Lecture Notes in Computer Science, vol. 1148 (Springer, Berlin, 1996), pp. 203–222. From the First ACM Workshop on Applied Computational Geometry
36. J. Shun, G.E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pp 135–146 (2013)
37. P.-N. Tan, M. Steinbach, V. Kumar (eds.), *Introduction to Data Mining* (Pearson Addison Wesley, Boston, 2005)
38. J.R. Ullmann, Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. J. Exp. Algorithm. **15**, 1.6:1.1–1.6:1.64 (2011) ISSN 1084-6654. doi:10.1145/1671970.1921702. http://doi.acm.org/10.1145/1671970.1921702
39. L.G. Valiant, A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). ISSN 0001-0782. doi:http://doi.acm.org/10.1145/79173.79181
40. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The splash-2 programs: characterization and methodological considerations, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95* (ACM, New York, 1995), pp. 24–36. ISBN 0-89791-698-0. doi:10.1145/223982.223990. http://doi.acm.org/10.1145/223982.223990

Chapter 3
Emerging Circuit Technologies: An Overview on the Next Generation of Circuits

Robert Wille, Krishnendu Chakrabarty, Rolf Drechsler, and Priyank Kalla

1 Introduction

In the last decades, great progress has been made in the development of computing machines resulting in electronic systems which can be found in almost every aspect of our daily life. All this has become possible due to the achievements made in the domain of semiconductors which is usually associated with *Moore's Law*—the famous prediction by Gordon Moore that the number of transistors in an electronic device doubles every 18 months. While this prediction is still holding on, physical boundaries and cost restrictions of conventional CMOS-based circuitry led to an increasing interest in alternative technologies (so-called *More than Moore* technologies). Besides that, the advances according to Moore's Law

R. Wille (✉)
Institute for Integrated Circuits, Johannes Kepler University Linz, Linz,
Austria Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: robert.wille@jku.at

K. Chakrabarty
Duke University, Durham, NC, USA
e-mail: krish@duke.edu

R. Drechsler
Group for Computer Architecture, University of Bremen, Bremen, Germany

Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: drechsler@uni-bremen.de

P. Kalla
Department of Electrical and Computer Engineering, University of Utah,
Salt Lake City, UT, USA
e-mail: kalla@ece.utah.edu

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_3

also lead to the consideration of application areas for electronic systems which go beyond just performing computations and complement the digital part by non-digital functionality (leading to so-called *More than Moore* technologies).

For both directions, various technologies are currently considered. Many of them are still in a rather academic state; others already find practical and commercial application. All of them however rely on significantly different paradigms than established (CMOS-based) circuit technologies. In this chapter, we provide a brief overview on *selected* emerging technologies, namely *Digital Microfluidic Biochips*, *Integrated Photonic Circuits*, and *Reversible Circuits*. For each technology, we review the respective background and outline corresponding application areas. Afterwards, main challenges for the design of these circuits are discussed.

2 Digital Microfluidic Biochips

According to a recent announcement by Illumina, a market leader in DNA sequencing, digital microfluidics has been transitioned to the marketplace for sample preparation [44]. This technology has also been deployed by Genmark for infectious disease testing [26] and FDA approval is expected soon for an analyzer from Baebies to detect lysosomal enzymes in newborns [23]. These milestones highlight the emergence of *Digital Microfluidic Biochip* (DMFB) technology for commercial exploitation and its potential for point-of-care diagnosis [82], sample processing [78], and cell-based assays [11]. Using DMFBs, bioassay protocols are scaled down to droplet size and executed by program-based control of nanoliter droplets on a patterned array of electrodes. The integration of sensors and imaging techniques on this platform led to the first generation of cyberphysical DMFBs [56], and software-based dynamic adaptation in response to sensor feedback has been utilized for error recovery [1, 47, 58]. Due to the fundamental importance of genomic analysis, considerable effort has similarly been devoted to the design and implementation of miniaturized platforms for gene-expression analysis [38, 54, 66, 73].

2.1 Technology Platform

A DMFB utilizes electrowetting-on-dielectric to manipulate and move microliter or nanoliter or picoliter droplets containing biological samples on a two-dimensional electrode array [82]. A unit cell in the array includes a pair of electrodes that acts as two parallel plates. The bottom plate contains a patterned array of individually controlled electrodes, and the top plate is coated with a continuous ground electrode. A droplet rests on a hydrophobic surface over an electrode, as shown in Fig. 3.1. It is moved by applying a control voltage to an electrode adjacent to the droplet and, at the same time, deactivating the electrode just under the droplet. Using interfacial tension gradients, droplets can be moved to any location on a two-dimensional array.

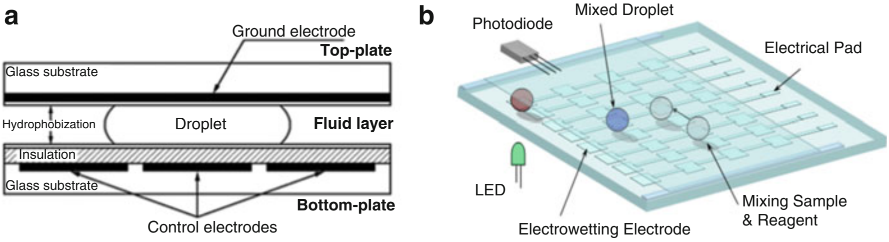


Fig. 3.1 Schematic of a digital microfluidic biochip. (**a**) Basic unit cell. (**b**) Top view

A film of silicone oil is used as a filler medium to prevent cross contamination and evaporation. This approach allows for rapid, massively parallel processing of substances without the restriction of a flow path defined at fabrication time. In addition to electrodes, optical detectors and capacitive sensors have been integrated in digital microfluidic arrays. This platform consumes less than 1 μW of power per droplet operation, therefore a million operations with nanoliter/picoliter droplets can be simultaneously carried out at less than 1 W power.

Videos of microfluidics in action are available at http://microfluidics.ee.duke.edu. Digital microfluidics works much the same way as traditional benchtop protocols, only with much smaller volumes and much higher automation. A wide range of established chemistries and protocols can be seamlessly transferred to a droplet format.

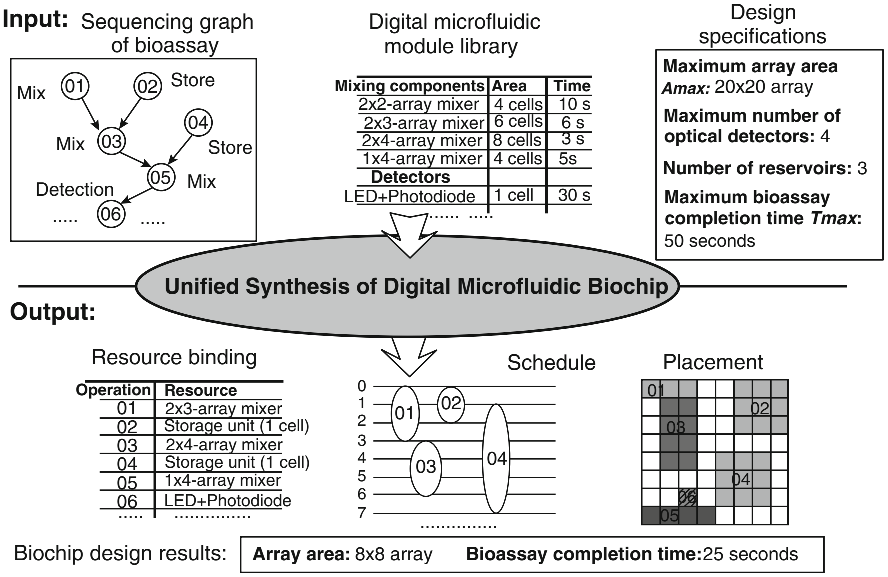
2.2 Design Methods for DMFBs: Today's Solutions

Until recently, research on design automation for DMFBs focused exclusively on scheduling, resource binding, droplet routing, and mapping of control pins to electrodes [15, 29–31, 39, 46–48, 52, 59, 86–90, 108–114]. Droplet-routing methods have also been developed to avoid cross-contamination [116, 117]. These methods can reduce droplet transportation time by finding optimal routing plans. Synthesis methods that combine defect-tolerant architectural synthesis with droplet-routing-aware physical design have also been developed. Droplet routability, defined as the ease with which pathways can be determined, has been estimated and integrated in the synthesis flow. Figure 3.2 illustrates high-level synthesis for DMFBs.

The first demonstrations of the interplay between hardware and software in the biochip platform were presented in [36, 37]. Videos are available at:

- http://microfluidics.ee.duke.edu/Published_Videos/2013_DATE/
- http://microfluidics.ee.duke.edu/BioCAS2015_IntegratedErrorRecovery/

These demos highlight autonomous cyberphysical operation without human intervention. The control of multiple droplets in a fabricated biochip by a pre-



Mixing components	Area	Time
2x2-array mixer	4 cells	10 s
2x3-array mixer	6 cells	6 s
2x4-array mixer	8 cells	3 s
1x4-array mixer	4 cells	5s
Detectors		
LED+Photodiode	1 cell	30 s
....		

Maximum array area ***Amax:*** 20x20 array

Maximum number of optical detectors: 4

Number of reservoirs: 3

Maximum bioassay completion time ***Tmax*****:** 50 seconds

Operation	Resource
01	2x3-array mixer
02	Storage unit (1 cell)
03	2x4-array mixer
04	Storage unit (1 cell)
05	1x4-array mixer
06	LED+Photodiode
.....	....

Array area: 8x8 array **Bioassay completion time:**25 seconds

Fig. 3.2 Illustration of automated synthesis of DMFBs

programmed assay plan and dynamic adaptation has been accomplished based on feedback from capacitance sensing. Algorithms for sample preparation [35, 64, 75], hardware/software co-design for lab-on-chip [57], and optimization techniques for protocols such as PCR [51, 58] have also been developed.

2.3 *Design Methods for DMFBs: Looking Ahead*

Quantification of the expression level for a gene is a widely used technique in molecular biology [55]. An important application of this technique is in epigenetics, which identifies changes in the regulation of gene expression that are not dependent on gene sequence. Often, these changes occur in response to the way the gene is packaged into chromatin in the nucleus. For example, a gene can be unfolded ("expressed"), be completely condensed ("silenced"), or be somewhere in between. Each distinct state is characterized by chromatin modifications that affect gene behavior [9]. An improved understanding of the in vivo cellular and molecular pathways that govern epigenetic changes is needed to define how this process alters gene function and contributes to human diseases [107].

A key application of DMFBs lies in quantitative biomolecular analysis, with applications to epigenetics. However, a significant rethinking in system design is needed to ensure dynamic adaptation for quantitative analysis on-chip. Figure 3.3 illustrates a 5-layer C5 (based on the C for each level) architecture that has recently been proposed [41]. Today's design methods incorporate online biochemistry-on-

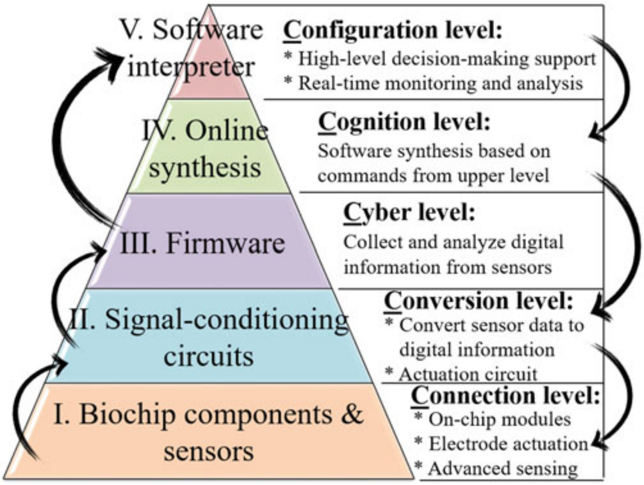

V. Software interpreter
IV. Online synthesis
III. Firmware
II. Signal-conditioning circuits
I. Biochip components & sensors
Configuration level:
* High-level decision-making support
* Real-time monitoring and analysis
Cognition level:
Software synthesis based on commands from upper level
Cyber level:
Collect and analyze digital information from sensors
Conversion level:
* Convert sensor data to digital information
* Actuation circuit
Connection level:
* On-chip modules
* Electrode actuation
* Advanced sensing

Fig. 3.3 The 5-layer (C^5) architecture for DMFBs [41]

chip synthesis, which provides reconfiguration capability to recover from operational errors (Level IV in Fig. 3.3) [1, 45, 56]. However, there is still a gap between the physical space and online synthesis, which impedes the use of DMFBs for quantitative analysis due to the lack of autonomous data analysis and intelligent decision-making. We will explore algorithmic innovations that fill the gap between the control and monitoring of the physical space on one side, and the cyber space (i.e., online biochemistry-on-chip synthesis) on the other side. Coupling the firmware with online synthesis will potentially open new opportunities for dynamic synthesis. For example, the need for short time-to-result might require the prioritization and selection of samples for detection. These ideas can also be extended to prioritize bioassays for synthesis in a multi-assay setting. Such coupling between the firmware and online synthesis is important in type-driven single-cell analysis [34], where the selection of a quantitative protocol depends on the cell type.

Next-generation design methods must integrate these levels to enable the seamless on-chip execution of complex biochemical protocols. As shown in Fig. 3.3, each level is expected to play a distinct role. With the integration of sensors at the hardware level (Level I), there is a need to provide analog signal acquisition and digital signal processing capabilities to transform the received signals into readable data. This can be achieved via a signal conditioning level (Level II). Previous designs of cyberphysical DMFBs have attempted to integrate this level with the system infrastructure, but only for the limited purpose of error recovery [56]. The uppermost level (Level V) serves as the system coordinator. It is responsible for adapting the flow of protocol execution based on the decisions conveyed from the firmware (Level III). Adaptation is supported by an application model, which keeps track of application progress, and a performance model that keeps track of resource utilization. Despite the large body of published work on design automation for DMFBs, integrated hardware/software solutions for DMFBs thus far have been limited to on-chip droplet manipulation (either only Level IV of the proposed C5 architecture or the interplay between Level I and Level II). There is a need for more relevant design-automation techniques that can be used for biology-on-a-chip.

To enable quantitative applications on a DFMB, researchers have carried out a benchtop ("wet lab") laboratory study on gene-expression analysis that required reliable concurrent manipulation of independent samples, as well as sample-dependent decision-making [42]. The goal was to study the transcriptional profile of a *Green Fluorescent Protein* (GFP) reporter gene under epigenetic control [106, 107]. Control (GFP not under epigenetic control) and experimental strains were analyzed by qPCR (following cell lysis, mRNA isolation and purification, and cDNA synthesis). These studies allowed an assessment of the quality of the protocol, define the influence of epigenetic chromatin structures on gene expression, and provided guidance for the synthesis of the underlying protocol on to the chip.

With multiple samples and with several causative factors affecting chromatin behavior, implementing epigenetic-regulation analysis using a benchtop setting is tedious and error-prone; thus, this preliminary benchtop study motivates the need to miniaturize epigenetic-regulation analysis. The benchtop study also provided important guidance on the design of the miniaturized protocol for a DMFB. Figure 3.4a depicts a flowchart of the benchtop study. Efforts have been made to adapt this protocol for digital microfluidics, leading to the miniaturized protocol in Fig. 3.4b. Intermediate decision points have been used to depict the control of the protocol flow for every sample.

Based on the above experimental work, a design method for a DMFB that performs quantitative gene-expression analysis based on multiple sample pathways was recently presented [42, 43]. This design uses an experimentally validated electrode-degradation model, spatial reconfiguration, and adaptive shared-resource allocation in order to efficiently utilize on-chip modules. A time-wheel for run-time coordination, shown in Fig. 3.5, was developed in this work.

3 Integrated Photonic Circuits

Silicon-based integrated optics, dubbed *Si-photonics* [85], offers the potential for low-latency, low-power, and high-bandwidth interconnect solutions for on-chip communications fabric [3, 4, 14]. In such a technology, optical signals are coupled to silicon waveguides and routed across the chip by modulating on-chip or off-chip lasers. Modulation can be performed by means of interferometry, using *Mach-Zehnder Interferometers* (MZIs) [53], or by means of resonance, using *Ring Resonators* (RRs) [10]. An electrical signal modulates the laser, essentially transforming electrical data into optical signals. The various applications of Si-photonics have been made viable due to a number of recent breakthroughs [12, 28, 53, 74], that also exploit the already available silicon-based fabrication infrastructure [68] and design technology [20].

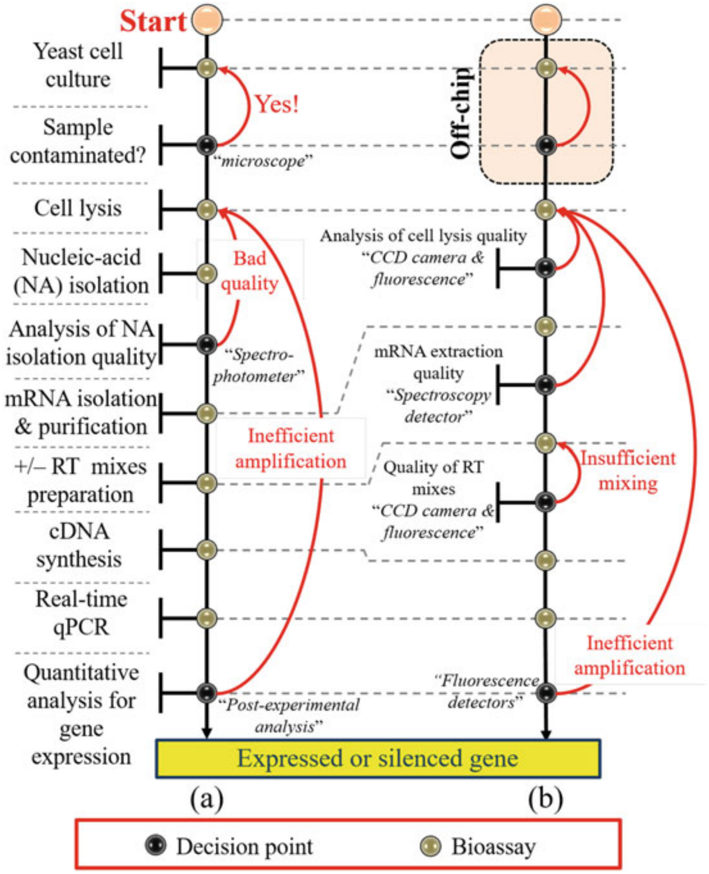

Start
Yeast cell culture
Sample contaminated?
Yes!
"microscope"
Cell lysis
Nucleic-acid (NA) isolation
Bad quality
Analysis of NA isolation quality
"Spectro-photometer"
mRNA isolation & purification
Inefficient amplification
+/– RT mixes preparation
cDNA synthesis
Real-time qPCR
Quantitative analysis for gene expression
"Post-experimental analysis"
Off-chip
Analysis of cell lysis quality
"CCD camera & fluorescence"
mRNA extraction quality
"Spectroscopy detector"
Quality of RT mixes
"CCD camera & fluorescence"
Insufficient mixing
Inefficient amplification
"Fluorescence detectors"
Expressed or silenced gene
(a)
(b)
● Decision point
● Bioassay

Fig. 3.4 Quantitative analysis of gene expression: (**a**) bench-top setup (**b**) DMFB-based

Advancements in Si-photonics are also enabling investigations into applications beyond telecom: sensing, filtering, quantum technology [71]—and even optical computing [13, 17]. By abstraction and modeling of MZI or RR based modulators as switches that are interconnected with waveguides, it has become possible to realize logic circuits in opto-electronics. In effect, we are now seeing a *convergence of communication and computation*, which provides the opportunity to embed logic and computation within the communications fabric. This can significantly extending the realm of applications that can be served by integrated optics. In this section, we review the design of optical logic circuits in Si-photonics based linear optical technologies, and address the challenges of on-chip integration in the presence of thermal-gradients across the optical substrate.

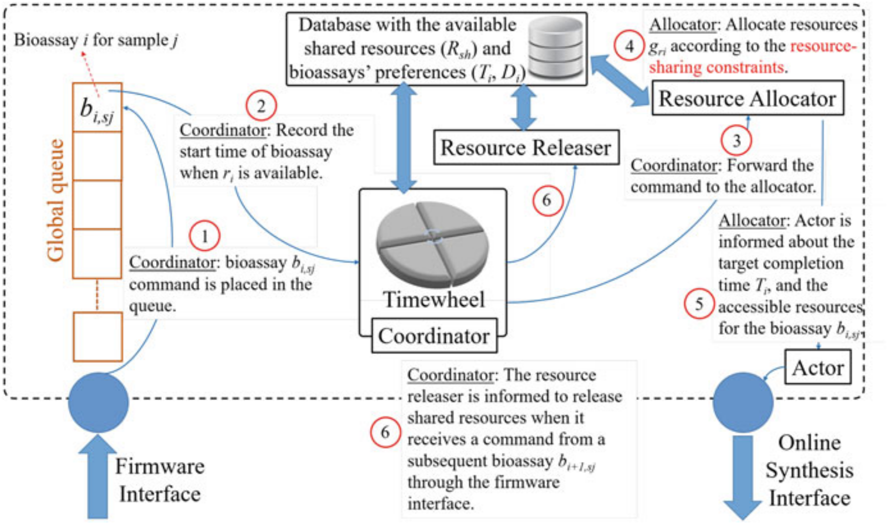

Bioassay i for sample j
$b_{i,sj}$
Global queue
Database with the available shared resources (R_{sh}) and bioassays' preferences (T_i, D_i)
2
Coordinator: Record the start time of bioassay when r_i is available.
1
Coordinator: bioassay $b_{i,sj}$ command is placed in the queue.
Resource Releaser
Timewheel
Coordinator
6
4
Allocator: Allocate resources g_{ri} according to the resource-sharing constraints.
Resource Allocator
3
Coordinator: Forward the command to the allocator.
Allocator: Actor is informed about the target completion time T_i, and the accessible resources for the bioassay $b_{i,sj}$
5
Actor
Coordinator: The resource releaser is informed to release shared resources when it receives a command from a subsequent bioassay $b_{i+1,sj}$ through the firmware interface.
6
Firmware Interface
Online Synthesis Interface

Fig. 3.5 Shared-resource coordination framework [42]

3.1 Photonic Logic Circuit Model

To enable logic design and scalable synthesis methodologies, a key aspect is to use conventional routing devices—specifically crossbar routing devices—as the *building-blocks* for logic design. Integrated optical devices transmit and route light using optical waveguides [67], which are created as guiding layers on the substrate using lithographic and deposition methods. In contemporary systems, light is coupled into the system from the outside using a laser, and sensed, at the destination, by optical receivers using a light-detection material such as germanium. We describe our basic optical logic element using the MZI (the model can be analogously described using the RRs). The MZI is a conventional integrated optic device found in many designs, used for modulation but also routing through the use of coupling and controlled interference.

Consider the MZI depicted in Fig. 3.6a, with inputs P and Q, and outputs F and G. Between P and F as well as Q and G are waveguides, with an index of refraction n. Coupling occurs when two waveguides are brought within in close proximity to each other such that the electromagnetic fields in one waveguide extend over the other waveguide and vice versa, causing energy to cross over between one waveguide to the other, as a function of coupling length. The couplers in this device are 3dB couplers, tuned to divide and/or combine the signal from both inputs equally between the two outputs. In Fig. 3.6a, the signal at (a) passes through the 3dB coupler and is divided between the outputs (b) and (c), inducing a $\pi/2$ phase change in (c).

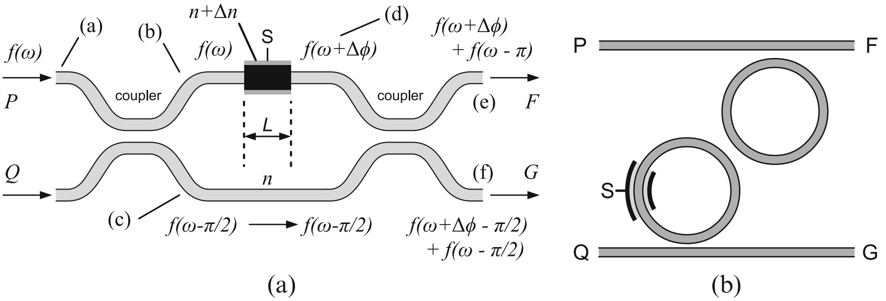


Fig. 3.6 Optical modulator devices. (**a**) Mach-Zehnder Interferometer. (**b**) Ring Resonator Modulator

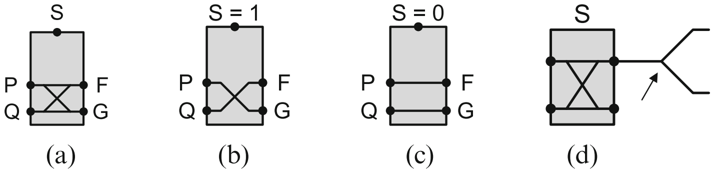


Fig. 3.7 Cross-bar switch model. (**a**) Gate. (**b**) Cross. (**c**) Bar. (**d**) Splitter

In the center region, S is an outside input used to affect the refractive index of the upper waveguide by Δn by using methods/devices such as microheaters, carrier injection, advanced methods such as high-speed MOS-capacitors [53], or other means. This change in refractive index causes a path-length difference, and therefore a phase difference, between the signals in (b) and (d). This phase difference causes constructive or destructive interference at the second coupler when the signals from (c) and (d) are combined. A phase difference of 0 or π will route each input completely to one output or the other, and the device acts as the controlled crossbar depicted in Fig. 3.7a. Similar operation can also be achieved by using ring resonator modulators shown in Fig. 3.6b.

The operation of the MZI allows us to model it as a crossbar gate that routes light signal completely between two paths depending on the state of S, and depict it symbolically in Fig. 3.7a, with its two states in Fig. 3.7b, c (bar and cross respectively). The waveguides are sourced by light (logical "1") or darkness ("0"), and the output of a function is read using optical receivers at the end. In our model, the switching input S is an electrical signal; it is an outside signal that controls the cross/bar configuration and cannot be switched by optical inputs. Light is assumed to move from the P and Q side to F and G. In our model, an optical signal cannot directly switch a crossbar's S input. More formally:

$$(S = 0) \implies (P = F) \wedge (Q = G); \;\; (S = 1) \implies (Q = F) \wedge (P = G).$$

These constraints affect how functions may be composed, and imply that the control inputs to a crossbar are the primary inputs for that network. In addition to MZIs, we also utilize optical splitters, depicted symbolically in Fig. 3.7d. A splitter divides the light from one waveguide into two output waveguides, each of which contain the original signal, but at half the power (a 3 dB loss). In our model, splitters are a signal degradation mechanism for a given topology, whereas we assume that waveguide lengths are essentially lossless. Insertion losses for MZI devices can be estimated to around 1 dB per device. Such losses can be factored into heuristics once physical layout information is available; interested reader may refer to [18] for more details on loss-constrained layout synthesis. All designs created using the above model can be physically realized, including allowing waveguides to cross each other without interference.

3.2 Design and Synthesis of Photonic Logic Circuits

We introduce the concept of *Virtual Gates*, a scalable methodology for implementing Boolean functions as a network of nested templates constructed from interconnected MZI gates. We explore how these may be used directly, and how their limitations necessitate further research into techniques for logic sharing without violating the opto-electrical barrier.

A *Virtual Gate* (VG) is a crossbar gate that is switched by a function, not necessarily a primary input. The gate is "virtual" in the sense that it is a black box for a function composed of "real" gates—those driven by primary inputs—as well as other virtual gates. A novel form of nesting can be used to compose VG function implementations, where Boolean operators are implemented by replacing child gates with other gates that may be real or virtual.

A given VG implementation comprises two input waveguide ports p and q connected by waveguides and crossbar gates to two output ports f and g. The nesting operation comprises the Boolean operator forms depicted in Fig. 3.8, and is illustrated in Fig. 3.9 where two AND virtual gates are nested within an OR virtual gate, creating the final function $ab + cd$. Evaluation of a VG, given a primary input

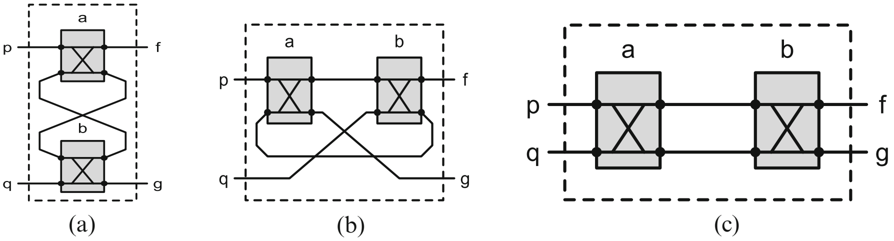


Fig. 3.8 Boolean operators in the virtual gate model. (**a**) *AND* (a, b). (**b**) *OR* (a, b). (**c**) *XOR* (a, b)

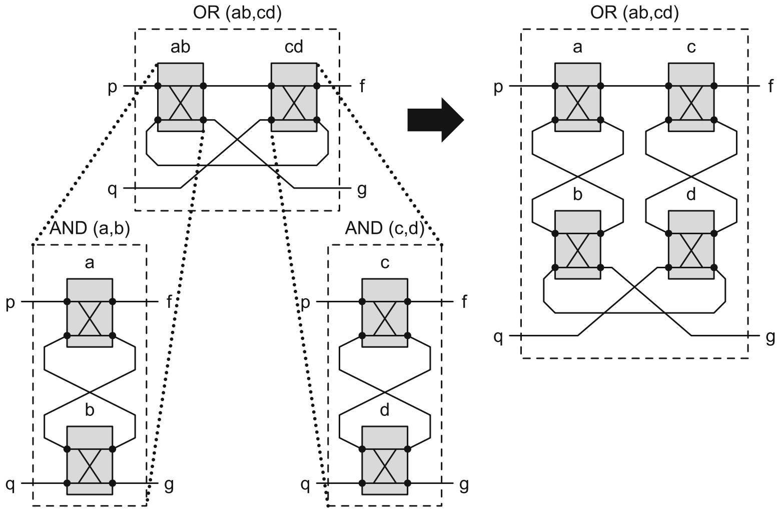


Fig. 3.9 Composing more complex functions using virtual gates

assignment, involves assigning p and q inputs logical 0 and 1, respectively, and applying cross or bar configurations to gates as defined in Fig. 3.7. The output of the function is detected at f, with $g = \neg f$ simultaneously providing its complement.

The major limitation of designing with virtual gates is that the nesting of gates prevents the extraction/sharing of arbitrary *Common Sub-Expressions* (CSE). For example, in Fig. 3.10 one cannot simply share the ab term from $f = ab + cd$ for use with another gate; assignments such as $abcd = \{1; 1; 1; 1\}$ will cause all crossbar gates to assume a cross-configuration, isolating the top input of the h-gate from the optical inputs of the network. In effect, any operator employing feedback for its inputs can produce an undefined state. Only the XOR operator does not exhibit this behavior as it has no feedback (see Fig. 3.8c), but XOR-based CSE is not well studied in contemporary logic synthesis. To address this issue particularly for optical logic synthesis, we investigated a XOR-based functional decomposition technique for CSE, and implemented it within our virtual-gate paradigm; interested reader can refer to [17] for more details. Despite recent efforts [21, 103], the *minimal-loss multi-level optical logic synthesis problem* remains to be satisfactorily solved. As every MZI (or RR) device incurs insertion loss (approx. 1dB/device), techniques to reduce the literal-count (or device-count) need to be employed. One such approach is CSE extraction. However, this requires the use of splitters, which in turn introduces losses due to signal sharing (approx. 3db/splitter). These seemingly contradictory requirements of minimizing both the device and splitter count need to be accounted for more effectively in optical logic synthesis techniques.

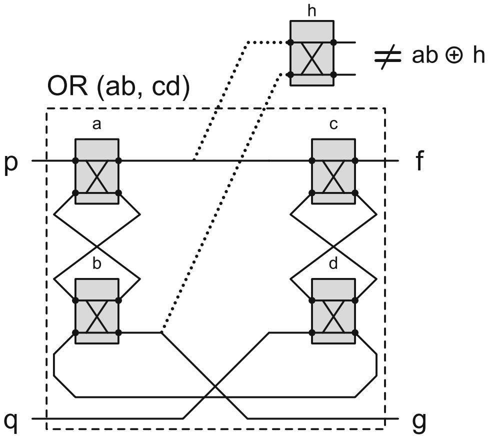


Fig. 3.10 Limitations of virtual gates in common subexpression sharing

3.3 Challenges of Si-Photonic Integration: Thermal Issues

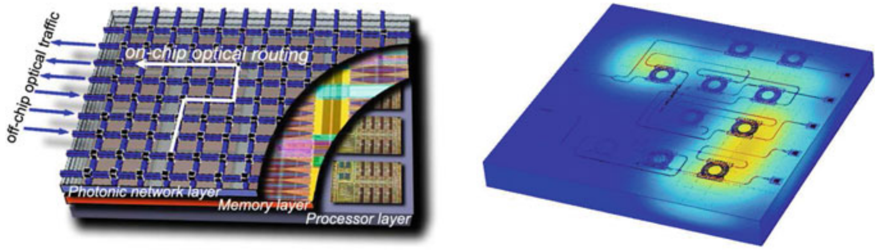
Si-photonics also finds applications in the design of *Optical Networks-on-Chip* (ONoC) architectures [3, 4, 16, 72]. Figure 3.11a depicts a typical 3D-stacking scenario where an ONoC is integrated with a multicore processor system. The ONoC consists of wavelength-division multiplexed waveguides and modulators comprising RRs or MZIs, that route packets of data across the chip in the optical domain. On-chip integration of *Opto-Electronic Integrated Circuits* (OEICs) poses the problem of *thermal-aware* design and synthesis. Si-photonic modulators such as the MZIs and RRs lie at the core of photonic network systems. Such OEICs are extremely sensitive to temperature-induced changes in refractive index.

For example, referring to Fig. 3.6a, the routing controlled by input S can be described by the following equations:

$$\phi_1 = \frac{\omega}{c} \cdot n \cdot L \quad \phi_2 = \frac{\omega}{c} \cdot (n + \Delta n) \cdot L \tag{3.1}$$

$$\Delta\phi = |\phi_2 - \phi_1| = \pi = \frac{\omega}{c} \cdot \Delta n \cdot L \tag{3.2}$$

where ω is the angular frequency of the light (dependent on wavelength), ϕ_1 and ϕ_2 represents the phase of the light in the two center waveguides, and n is the index of refraction for the waveguide. Due to silicon's large thermo-optic coefficient of $\frac{\Delta n}{\Delta T} = 1.86 \times 10^{-4}/°\mathrm{K}$, a change in temperature (ΔT) will cause a change in $\Delta\phi$, thus interfering with modulator operation. Particularly with regards to the system shown in Fig. 3.11a, the underlying electronic layer (computational units/cores) will act as heat sources within the chip. These temperature hot-spots will generate a thermal gradient across the optical routing substrate. This, in turn, will cause temperature-induced changes to the refractive index of the photonic layer, making the system error-prone and unworkable. Furthermore, the locality of heat sources means that



(a) (b)

Fig. 3.11 Optical network-on-chip and thermal issues. (**a**) Hybrid opto-electronic integration for optical network-on-chip. *Image credit: IBM [40].* (**b**) On-chip heat sources creating thermal gradients across optical substrate

different modulators will be subjected to different temperature conditions such as depicted in Fig. 3.11b. Such external thermal gradients pose significant operational challenges in hybrid opto-electronic integration.

The photonic design and automation communities have begun to address remedial measures for such thermal-sensitive systems. Such measures entail some form of active or passive *compensation* (tuning) to mitigate the effect of refractive index variations. Active tuning utilizes external effects to change the optical properties of materials. Active tuning is usually implemented via microheaters [24] embedded around the waveguides or overlaid on the oxide cladding, to effect a temperature-induced compensation in the refractive index. Tuning is also achieved by using carrier injection, by applying a DC-bias to the modulator [80]. Highly doped P and N regions are used to form a P-i-N junction around the modulator waveguides, and free-carriers are injected into (conversely, extracted from) the devices to cause carrier concentration induced refractive index changes. Operating system scheduling techniques have also been proposed to control dynamic compensation [72].

Such techniques are costly in terms of power, area, and restrictive in terms of tuning range. Microheater-based tuning is very slow, cumbersome to tune precisely, consumes large amount of power, and it exacerbates the already strained energy-density on chip. On the other hand, DC-bias based tuning has limited range. To address these limitations, [19] proposes a template-based RR design that enables process-compatible *(re)synthesis* to compensate for pre-computed average- or worst-case thermal gradients. One can assume that an average-case or worst-case workload characterization is known for the multiprocessor system, from which the thermal characterization over the optical chip can be determined utilizing techniques such as in [77]. Based on this data, device (re)synthesis techniques are proposed in [19] that may enable combined static and active tuning approaches for thermal-aware synthesis of photonic layers.

In the context of optical computing and hybrid opto-electronic integration, thermal-awareness in design and automation is an imperative. The approaches discussed above are in their infancy and they have yet to be validated on large-scale systems. Further research is needed to fully exploit the full potential of Si-photonics in computing systems.

4 Reversible Circuits: A Basis for Quantum Computation, Encoder Design, and More

In reversible circuits, functionality is specified and, eventually, realized in a bijective fashion, i.e. a unique input/output mapping is enforced. Their main difference to conventional circuits can already be illustrated by a simple standard operation such as the logical AND gate. While it is possible to obtain the inputs of an AND gate if the output signal is 1 (then, both inputs must be assigned the value 1 as well), it is not possible to unambiguously determine the input values if the AND outputs 0. In contrast, reversible circuits ought to allow computations in both directions, i.e. the inputs can be obtained from the outputs and vice versa.

4.1 Circuit Model

A *reversible circuit* G is a cascade of reversible gates, where fanout and feedback are not directly allowed [65]. Each variable of the function f is represented by a *circuit line*, i.e. a signal through the whole cascade structure on which the respective computation is performed. Computations are performed by *reversible gates*. In the literature, reversible circuits composed of *Toffoli gates* are frequently used. A Toffoli gate is composed of a (possibly empty) set of *control lines* $C = \{x_{i_1}, \ldots, x_{i_k}\} \subset X$ and a single *target line* $x_j \in X \setminus C$. The Toffoli gate inverts the value on the target line if all values on the control lines are assigned to 1 or if $C = \emptyset$, respectively. All remaining values are passed through unaltered.

Figure 3.12a shows a Toffoli gate drawn in standard notation, i.e. control lines are denoted by ●, while the target line is denoted by ⊕. A circuit composed of several Toffoli gates is depicted in Fig. 3.12b. This circuit maps, e.g., the input 111 to the output 110 and vice versa.

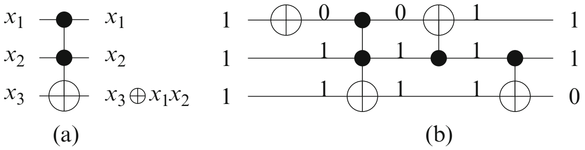


Fig. 3.12 Reversible circuit. (**a**) Toffoli gate. (**b**) Toffoli circuit

4.2 Application Areas

The characteristic of being reversible is perfectly in line with requirements for many application areas, e.g. quantum computations or encoders inherently rely on such one-to-one mappings. Besides that, certain characteristics exist which are also interesting for low power design.

4.2.1 Quantum Computation

In a quantum computer [65], information is represented in terms of *qubits* instead of bits. In contrast to Boolean logic, qubits do not only allow to represent Boolean 0's and Boolean 1's, but also the superposition of both. In other words, using quantum computation and qubits in superposition, functions can be evaluated with different possible input assignments in parallel. Unfortunately, it is not possible to obtain the current state of a qubit. Instead, if a qubit is measured, either 0 or 1 is returned depending on a respective probability.

Nevertheless, using these quantum mechanical phenomena, quantum computation allows for breaching complexity bounds which are valid for computing devices based on conventional mechanics. The Grover search [33] and the factorization algorithm by Shor [81] rank among the most famous examples for quantum algorithms that solve problems in time complexities which cannot be achieved using conventional computing. The first algorithm addresses thereby the search of an item in an unsorted database with k items in time $O(\sqrt{k})$, whereas conventional methods cannot be performed using less than linear time. Shor's algorithm performs prime factorization in polynomial time, i.e. the algorithm is exponentially faster than its best known conventional counterpart. First physical realizations of quantum circuits have been presented, e.g., in [92].

Reversible circuits are of interest in this domain since all quantum operations inherently are reversible. Since most of the known quantum algorithms include a large Boolean component (e.g., the database in Grover's search algorithm and the modulo exponentiation in Shor's algorithm), the design of these components is often conducted by a two-stage approach, i.e. (1) realizing the desired functionality as a reversible circuit and (2) map the resulting circuit to a functionally equivalent quantum circuit (using methods introduced, e.g., in [2, 63]).

4.2.2 Encoder Design

Encoding devices represent a vital part of numerous applications realized in today's electronic systems such as addressing memories and caches, data demultiplexing, etc. (see [6, 70]). With the rise of *System on Chip* and *Network on Chip* architectures, they gained further importance by the fact that those architectures usually rely on a rather sophisticated interconnect solutions [5, 25, 50, 69].

However, their design significantly suffers from the fact that, using conventional design solutions, the respectively desired one-to-one mappings have to explicitly be specified. In its most straight forward fashion, this can be conducted by a complete specification of the mapping for each pattern—obviously leading to an exponential complexity. Even if the desired input/output mapping is specified for a (non-exponential) selection of the patterns only (e.g., the most important ones), conventional solutions still require to explicitly guarantee a valid one-to-one mapping for all remaining patterns.

Since reversible circuits inherently realize one-to-one mappings only, they provide a promising alternative to that. In fact, using reversible circuits, it is sufficient to only realize the mapping for the actually desired patterns. A valid one-to-one mapping for the remaining patterns is then inherently guaranteed by the reversibility of the circuit model. A solution where this concept is utilized can be found in [119]. Besides that, this principle has successfully been applied in the design of on-chip interconnects in [101, 104].

4.2.3 Low Power Design

Pioneering work by Landauer [49] showed that, regardless of the underlying technology, losing information during computation causes power dissipation. More precisely, for each “lost” bit of information, at least $k \cdot T \cdot \log(2)$ Joules are dissipated (where k is the Boltzmann constant and T is the temperature). Since today’s computing devices are usually built of elementary *gates* like AND, OR, NAND, etc., they are subject to this principle and, hence, dissipate this amount of power in each computational step.

Although the theoretical lower bound on power dissipation still does not constitute a significant fraction of the power consumption of current devices, it nonetheless poses an obstacle for the future. Figure 3.13 illustrates the development of the power consumption of an elementary computational step in recent and expected future CMOS generations (based on values from [115]). The figure shows that today’s technology is still a factor of 1000 away from the Landauer limit and that the expected CMOS development will reduce this to a factor of 100 within the next 10 years. However, a simple extrapolation also shows that the trend cannot continue with the current family of static CMOS gates as no amount of technological refinement can overcome the Landauer barrier. Moreover, the Landauer limit is only a *lower* bound on the dissipation. Gershenfeld has shown that the actual power dissipation corresponds to the amount of power used to represent a signal [27], i.e. Landauer’s barrier is closer than immediately implied by the extrapolation from Fig. 3.13.

Since reversible circuits bijectively transforms data at each computation step, the above-mentioned information loss and its resulting power dissipation does not occur. Because of this, reversible circuits manifest themselves as the only way to break this fundamental limit. In fact, it has been proven that to enable computations with no power consumption at all, the underlying realization must

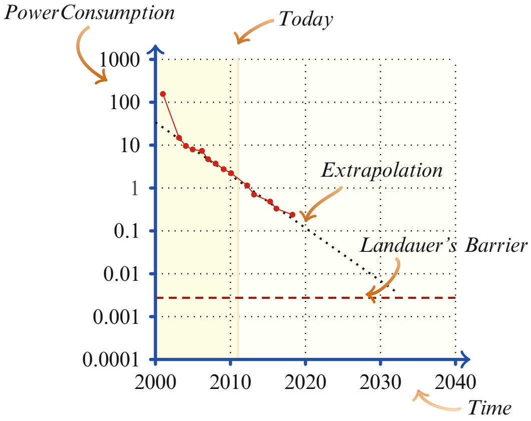


Fig. 3.13 Power consumption Q in different CMOS generations. The figure illustrates the development of the power consumption of an elementary computational step in recent CMOS generations (based on values from [115]). The power consumption is thereby determined by CV_t^2, where V_t is the threshold voltage of the transistors and C is the total capacitance of the capacitors in the logic gate. The capacitance C is directly proportional to $\frac{LW}{t}$, i.e. to the length L and the width W of the transistors. Reducing these sizes of transistors enables significant reductions in the power consumption as shown in the extrapolation. However, this development will reach a fundamental limit when power consumption is reduced to $k \cdot T \cdot \log(2)$ Joule

follow reversible computation principles [7]. These fundamental results motivate researchers in investigating this direction further. First physical realizations have been presented, e.g., in [8].

4.3 Design of Reversible Circuits

In order to exploit the potential of reversible computations in the application areas sketched above, an efficient design flow must be available. For conventional computation, an elaborated design flow emerged over the last 20–30 years. Here, a hierarchical flow composed of several abstraction levels (e.g., specification level, electronic system level, register transfer level, and gate level) supported by a wide range of modeling languages, system description languages, and hardware description languages has been developed and is in (industrial) use. In contrast, the design of circuits and systems following the reversible computation paradigm is still in its infancy (overviews can be found in [22, 76]). Although the basic tasks, i.e. synthesis, verification, and debugging, have been considered by researchers,[1] essential features and approaches of modern design flows are still missing. More precisely:

[1] A variety of corresponding open source implementations are available in the tool *RevKit* [83].

- Most of the existing approaches remain on the gate level. No real support of reversible circuits and systems on higher levels of abstractions are available.
- Most of the existing approaches for synthesis usually only accept specifications provided in terms of Boolean function descriptions like truth tables or Boolean decision diagrams (see, e.g., [32, 61, 79, 94]). Recently, hardware description languages for reversible circuits have been introduced, but support a very basic set of operations only (see, e.g., [91, 98, 105]).
- For verification and validation simulation engines (e.g., [93]) and equivalence checkers (e.g., [97]) are available so far. But the most efficient methods can handle circuits composed of at most 20,000 gates only (while approaches for conventional circuits are able to handle hundreds of thousands of gates).
- Debugging has hardly been considered [96, 99].

A main reason for these open issues surely is the different computation paradigm itself which requires that even the simplest operation has to be reversible. As a representative of a problem to overcome when discussing reversible rather than conventional computation devices is illustrated in the following by means of the adder function shown in Table 3.1a.

This adder has three inputs (the carry-bit c_{in} as well as the two summands x and y) and two outputs (the carry c_{out} and the *sum*). It surely belongs to one of the most important functions to be realized in terms of a circuit device. However, the adder obviously is not reversible (irreversible), since (1) the number of inputs differs from the number of outputs and (2) there is no unique input–output mapping. Even adding an additional output to the function (leading to the same number of input and outputs) would not make the function reversible. Then, the first four lines of the truth table can be embedded with respect to reversibility as shown in the rightmost column of Table 3.1a. However, since $c_{\text{out}} = 0$ and $sum = 1$ already appeared two times (marked bold), no unique embedding for the fifth truth table line is possible any longer. The same also holds for the lines marked italic.

This already has been observed in [60] and was further discussed in [100]. There, the authors showed that at least $\lceil \log(m) \rceil$ free outputs are required to make an irreversible function reversible, where m is the maximum number of times an output pattern is repeated in the truth table. Since for the adder at most 3 output pattern are repeated, $\lceil \log(3) \rceil = 2$ free outputs (and, hence, one additional circuit line) are required to make the function reversible.

Adding new lines causes constant inputs and garbage outputs. The value of the constant inputs can be chosen by the designer. Garbage outputs are by definition don't cares and thus can be left unspecified leading to an incompletely specified function. However, many synthesis approaches require a completely specified function so that all don't cares must be assigned with a concrete value.

As a result, the adder is embedded in a reversible function including four variables, one constant input, and two garbage outputs. A possible assignment to the constant as well as the don't care values is depicted in Table 3.1b. Note that the precise embedding may influence the respective synthesis results. Corresponding evaluations have been made, e.g., in [62, 95]. Moreover, determining an optimal

Table 3.1 Adder function and a possible embedding

(a) Adder function

c_{in}	x	y	c_{out}	sum	
0	0	0	0	0	0
0	0	1	**0**	**1**	**0**
0	1	0	**0**	**1**	**1**
0	1	1	*1*	*0*	*0*
1	0	0	**0**	**1**	**?**
1	0	1	*1*	*0*	*1*
1	1	0	*1*	*0*	*?*
1	1	1	1	1	1

(b) Embedding

0	$\mathbf{c_{in}}$	**x**	**y**	$\mathbf{c_{out}}$	**sum**	$\mathbf{g_1}$	$\mathbf{g_2}$
0	**0**	**0**	**0**	**0**	**0**	0	0
0	**0**	**0**	**1**	**0**	**1**	1	1
0	**0**	**1**	**0**	**0**	**1**	1	0
0	**0**	**1**	**1**	**1**	**0**	0	1
0	**1**	**0**	**0**	**0**	**1**	0	0
0	**1**	**0**	**1**	**1**	**0**	1	1
0	**1**	**1**	**0**	**1**	**0**	1	0
0	**1**	**1**	**1**	**1**	**1**	0	1
1	0	0	0	1	0	0	0
...							

embedding is a rather complex task—in fact has been proven to be coNP-complete in [84]. Recently, this complexity could have been tackled by an approach utilizing dedicated decision diagram techniques [118].

Besides that, alternative approaches (e.g., [94]) address this embedding problem not explicitly, but implicitly and lead to a significant amount of additional circuit lines. As this may harm in turn the efficiency of the resulting circuit (last but not least with respect to the power consumption in a possible application), how to trade-off these issues remains a big design challenge in this domain (see, e.g., [102]).

5 Conclusions

In this work, we provided an overview on the design of circuits for emerging technologies. Digital Microfluidic Biochips, Integrated Photonic Circuits, and Reversible Circuits are promising extensions and/or alternatives to conventional circuits and may revolutionize their respective application areas. At the same time, they work significantly different than their conventional counterpart and, hence, require dedicated design solutions. We reviewed the respective background and application areas and discussed the main design challenges to be addressed for each new generation of circuit technology.

References

1. M. Alistar, P. Pop, J. Madsen, Redundancy optimization for error recovery in digital microfluidic biochips. Des. Autom. Embed. Syst. **19**(1–2), 129–159 (2015)
2. A. Barenco, C.H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, H. Weinfurter, Elementary gates for quantum computation. Am. Phys. Soc. **52**, 3457–3467 (1995)
3. C. Batten, A. Joshi, V. Stojanovic, K. Asanovic, Designing chip-level nanophotonic interconnection networks. IEEE J. Emerging Sel. Top. Circuits Syst. **2**(2), 137–153 (2012). doi:10.1109/JETCAS.2012.2193932
4. R. Beausoleil et al., A nanophotonic interconnect for high-performance many-core computation, in *Symposium on High-Performance Interconnects*, pp. 182–189 (2008). doi:http://doi.ieeecomputersociety.org/10.1109/HOTI.2008.32
5. L. Benini, G.D. Micheli, E. Macii, M. Poncino, S. Quer, Power optimization of core-based systems by address bus encoding. IEEE Trans. VLSI Syst. **6**(4), 554–562 (1998). doi:10.1109/92.736127. http://dx.doi.org/10.1109/92.736127
6. L. Benini, G.D. Micheli, D. Sciuto, E. Macii, C. Silvano, Address bus encoding techniques for system-level power optimization, in *Design, Automation and Test in Europe*, pp. 861–866 (1998). doi:10.1109/DATE.1998.655959. http://dx.doi.org/10.1109/DATE.1998.655959
7. C.H. Bennett, Logical reversibility of computation. IBM J. Res. Dev **17**(6), 525–532 (1973)
8. A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, E. Lutz, Experimental verification of Landauer's principle linking information and thermodynamics. Nature **483**, 187–189 (2012)
9. C.R. Brown, C. Mao, E. Falkovskaia, M.S. Jurica, H. Boeger, Linking stochastic fluctuations in chromatin structure and gene expression. PLoS Biol. **11**(8), e1001621 (2013)
10. W. Bogaerts et al., Silicon microring resonators. Laser Photonics Rev. **6**(1), 47–73 (2012). doi:10.1002/lpor.201100017. http://dx.doi.org/10.1002/lpor.201100017
11. D. Bogojevic, M.D. Chamberlain, I. Barbulovic-Nad, A.R. Wheeler, A digital microfluidic method for multiplexed cell-based apoptosis assays. Lab Chip **12**(3), 627–634 (2012)
12. O. Boyraz, B. Jalali, Demonstration of a silicon Raman laser. Opt. Exp. **12**(21), 5269–5273 (2004). doi:10.1364/OPEX.12.005269. http://www.opticsexpress.org/abstract.cfm?URI=oe-12-21-5269
13. H.J. Caulfield et al., Generalized optical logic elements GOLEs. Opt. Commun. **271**, 365–376 (2007)
14. J. Chan, G. Hendry, K. Bergman, L.P. Carloni, Physical-layer modeling and system-level design of chip-scale photonic interconnection networks. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(10), 1507–1520 (2011). https://doi.org/10.1109/TCAD.2011.2157157
15. M. Cho, D.Z. Pan, A high-performance droplet routing algorithm for digital microfluidic biochips. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(10), 1714–1724 (2008)
16. M.J. Cianchetti, J.C. Kerekes, D.H. Albonesi, Phastlane: a rapid transit optical routing network, in *ACM SIGARCH Computer Architecture News*, vol. 37 (ACM, New York, 2009), pp. 441–450
17. C. Condrat, P. Kalla, S. Blair, Logic synthesis for integrated optics, in *Proceedings of the 21st Edition of the Great Lakes Symposium on Great Lakes Symposium on VLSI, GLSVLSI '11* (ACM, New York, 2011), pp. 13–18. doi:10.1145/1973009.1973013. http://doi.acm.org/10.1145/1973009.1973013
18. C. Condrat, P. Kalla, S. Blair, Crossing-aware channel routing for integrated optics. IEEE Trans. CAD of Integr. Circuits Syst. **33**(5), 814–825 (2014)
19. C. Condrat, P. Kalla, S. Blair, Thermal-aware synthesis of integrated photonic ring resonators, in *International Conference On Computer Aided Design (CAD)*, pp. 557–564 (2014)
20. C. Condrat, P. Kalla, S. Blair, More than moore technologies for next generation computer design, in *Design Automation for On-Chip Nanophotonic Integration* (Springer, New York, 2015), pp. 187–218

21. A. Deb, R. Wille, O. Keszöcze, S. Hillmich, R. Drechsler, Gates vs. splitters: contradictory optimization objectives in the synthesis of optical circuits. J. Emerg. Technol. Comput. Syst. **13**(1), 11 (2016)
22. R. Drechsler, R. Wille, From truth tables to programming languages: progress in the design of reversible circuits, in *International Symposium on Multi-Valued Logic*, pp. 78–85 (2011)
23. FDA News, FDA Advisors Back Approval of Baebies' Seeker Analyzer for Newborns (2016), [Online]. Available: http://www.fdanews.com/articles/178103-fda-advisors-back-approval-ofbaebies-seeker-analyzer-for-newborns
24. F. Gan, T. Barwicz, M. Popovic, M. Dahlem, C. Holzwarth, P.T. Rakich, H. Smith, E. Ippen, F. Kartner, Maximizing the thermo-optic tuning range of silicon photonic structures, in *Photonics in Switching, 2007*, pp. 67–68 (2007). doi:10.1109/PS.2007.4300747
25. A. García-Ortiz, D. Gregorek, C. Osewold, Optimization of interconnect architectures through coding: a review, in *Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International*, pp. 1–6 (2011). doi:10.1109/SIECPC.2011.5876688
26. GenMark Dx, GenMark ePlex System (2016), [Online]. Available: https://www.genmarkdx.com/solutions/systems/eplex-system/
27. N. Gershenfeld, Signal entropy and the thermodynamics of computation. IBM Syst. J. **35**(3–4), 577–586 (1996)
28. W. Green et al., Ultra-compact, low RF power, 10 Gb/s silicon Mach-Zehnder modulator. Opt. Exp. **15**(25), 17,106–17,113 (2007). http://www.opticsexpress.org/abstract.cfm?id=148351
29. E.J. Griffith, S. Akella, M.K. Goldberg, Performance characterization of a reconfigurable planar-array digital microfluidic system. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **25**(2), 340–352 (2006)
30. D. Grissom, P. Brisk, A field-programmable pin-constrained digital microfluidic biochip, in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 1–9 (2013)
31. D.T. Grissom, P. Brisk, Fast online synthesis of digital microfluidic biochips. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(3), 356–369 (2014)
32. D. Große, R. Wille, G.W. Dueck, R. Drechsler, Exact multiple control Toffoli network synthesis with SAT techniques. IEEE Trans. CAD **28**(5), 703–715 (2009)
33. L.K. Grover, A fast quantum mechanical algorithm for database search, in *Theory of Computing*, pp. 212–219 (1996)
34. S. Hosic, S.K. Murthy, A.N. Koppes, Microfluidic sample preparation for single cell analysis. Anal. Chem. **88**(1), 354–380 (2015)
35. Y.-L. Hsieh, T-Y. Ho, K. Chakrabarty, A reagent-saving mixing algorithm for preparing multiple-target biochemical samples using digital microfluidics. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **31**(11), 1656–1669 (2012)
36. K. Hu, B.-N. Hsu, A. Madison, K. Chakrabarty, R. Fair, Fault detection, real-time error recovery, and experimental demonstration for digital microfluidic biochips, in *Proceedings of IEEE/ACM Design, Automation and Test in Europe (DATE)*, pp. 559–564 (2013)
37. K. Hu, M. Ibrahim, L. Chen, Z. Li, K. Chakrabarty, R. Fair, Experimental demonstration of error recovery in an integrated cyberphysical digital-microfluidic platform, in *Proceedings of IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 1–4 (2015)
38. Z. Hua, J.L. Rouse, A.E. Eckhardt, V. Srinivasan, V.K. Pamula, W.A. Schell, J.L. Benton, T.G. Mitchell, M.G. Pollack, Multiplexed real-time polymerase chain reaction on a digital microfluidic platform. Anal. Chem. **82**(6), 2310–2316 (2010)
39. W. Hwang, F. Su, K. Chakrabarty, Automated design of pin-constrained digital microfluidic arrays for lab-on-a-chip applications, in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 925–930 (2006)
40. IBM Research, Silicon Integrated Nanophotonics Technology: From Lab to Fab (2012), http://www.research.ibm.com/photonics
41. M. Ibrahim and K. Chakrabarty, Cyberphysical adaptation in digitalmicrofluidic biochips, IEEE Biomedical Circuits and Systems Conference (BioCAS), Shanghai, 2016, pp. 444–447. doi: https://doi.org/10.1109/BioCAS.2016.7833827

42. M. Ibrahim, K. Chakrabarty, K. Scott, Synthesis of cyberphysical digital-microfluidic biochips for real-time quantitative analysis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **36**(5), 733–746 (2017). https://doi.org/10.1109/TCAD.2016.2600626
43. M. Ibrahim, K. Chakrabarty, K. Scott, Integrated and real-time quantitative analysis using cyberphysical digital-microfluidic biochips, in *Proceedings of IEEE/ACM Design, Automation and Test in Europe (DATE)*, pp. 630–635 (2016)
44. Illumina, Illumina NeoPrep Library Prep System (2015), [Online]. Available: http://www.illumina.com/systems/neoprep-library-system.html/
45. C. Jaress, P. Brisk, D. Grissom, Rapid online fault recovery for cyber-physical digital microfluidic biochips, in *Proceedings of IEEE VLSI Test Symposium (VTS)*, pp. 1–6 (2015)
46. O. Keszocze, R. Wille, R. Drechsler, Exact routing for digital microfluidic biochips with temporary blockages, in *International Conference on CAD*, pp. 405–410 (2014)
47. O. Keszocze, R. Wille, T.-Y. Ho, R. Drechsler, Exact one-pass synthesis of digital microfluidic biochips, in *Proceedings of the IEEE/ACM Design Automation Conference (DAC)* (2014)
48. O. Keszocze, R. Wille, K. Chakrabarty, R. Drechsler, A general and exact routing methodology for digital microfluidic biochips, in *International Conference on CAD*, pp. 874–881 (2015)
49. R. Landauer, Irreversibility and heat generation in the computing process. IBM J. Res. Dev. **5**, 183 (1961)
50. K. Lee, S. Lee, H. Yoo, Low-power network-on-chip for high-performance SOC design. IEEE Trans. VLSI Syst. **14**(2), 148–160 (2006). doi:10.1109/TVLSI.2005.863753. http://dx.doi.org/10.1109/TVLSI.2005.863753
51. Z. Li, T.-Y. Ho, K. Chakrabarty, Optimization of 3D digital microfluidic biochips for the multiplexed polymerase chain reaction. ACM Trans. Des. Autom. Electron. Syst. **21**(2), Article 25 (2016)
52. C. Liao, S. Hu, Physical-level synthesis for digital lab-on-a-chip considering variation, contamination, and defect. IEEE Trans. NanoBiosci. **13**(1), 3–11 (2014)
53. L. Liao et al., High speed silicon Mach-Zehnder modulator. Opt. Exp. **13**(8), 3129–3135 (2005). http://www.opticsexpress.org/abstract.cfm?URI=OPEX-13-8-3129
54. P. Liu, X. Li, S.A. Greenspoon, J.R. Scherer, R.A. Mathies, Integrated DNA purification, PCR, sample cleanup, and capillary electrophoresis microchip for forensic human identification. Lab Chip **11**(6), 1041–1048 (2011)
55. J. Lovén, D.A. Orlando, A.A. Sigova, C.Y. Lin, P.B. Rahl, C.B. Burge, D.L. Levens, T.I. Lee, R.A. Young, Revisiting global gene expression analysis. Cell **151**(3), 476–482 (2012)
56. Y. Luo, K. Chakrabarty, T.-Y. Ho, Error recovery in cyberphysical digital microfluidic biochips. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(1), 59–72 (2013)
57. Y. Luo, K. Chakrabarty, T.-Y. Ho, *Hardware/Software Co-Design and Optimization for Cyberphysical Integration in Digital Microfluidic Biochips* (Springer, Dordrecht, 2014)
58. Y. Luo, B.B. Bhattacharya, T.-Y. Ho, K. Chakrabarty, Design and optimization of a cyberphysical digital-microfluidic biochip for the polymerase chain reaction. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(1), 29–42 (2015)
59. E. Maftei, P. Pop, J. Madsen, Tabu search-based synthesis of dynamically reconfigurable digital microfluidic biochips, in *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 195–204 (2009)
60. D. Maslov, G.W. Dueck, Reversible cascades with minimal garbage. IEEE Trans. CAD **23**(11), 1497–1509 (2004)
61. D.M. Miller, D. Maslov, G.W. Dueck, A transformation based algorithm for reversible logic synthesis, in *Design Automation Conference*, pp. 318–323 (2003)
62. D.M. Miller, R. Wille, G. Dueck, Synthesizing reversible circuits for irreversible functions, in *EUROMICRO Symposium on Digital System Design*, pp. 749–756 (2009)
63. D.M. Miller, R. Wille, Z. Sasanian, Elementary quantum gate realizations for multiple-control Toffolli gates, in *International Symposium on Multi-Valued Logic*, pp. 288–293 (2011)
64. D. Mitra, S. Roy, S. Bhattacharjee, K. Chakrabarty, B.B. Bhattacharya, On-chip sample preparation for multiple targets using digital microfluidics. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(8), 1131–1144 (2014)

65. M. Nielsen, I. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, 2000)
66. H. Norian, R.M. Field, I. Kymissis, K.L. Shepard, An integrated CMOS quantitativepolymerase- chain-reaction lab-on-chip for point-of-care diagnostics. Lab Chip **14**(20), 4076–4084 (2014)
67. K. Okamoto, *Fundamentals of Optical Waveguides* (Academic, New York, 2000)
68. OpSIS, Optoelectronic System Integration in Silicon. http://www.opsisfoundry.org
69. A.G. Ortiz, L.S. Indrusiak, T. Murgan, M. Glesner, Low-power coding for networks-on-chip with virtual channels. J. Low Power Electron. **5**(1), 77–84 (2009). doi:10.1166/jolpe.2009.1006. http://dx.doi.org/10.1166/jolpe.2009.1006
70. P.R. Panda, N.D. Dutt, Reducing address bus transitions for low power memory mapping, in *1996 European Design and Test Conference, ED&TC 1996*, Paris, March 11–14, 1996, pp. 63–71 (1996). doi:10.1109/EDTC.1996.494129. http://dx.doi.org/10.1109/EDTC.1996.494129
71. A. Politi, J. Matthews, J. O'Brien, Shor's quantum factoring algorithm on a photonic chip. Science **325**, 1221 (2009)
72. A. Qouneh, Z. Li, M. Joshi, W. Zhang, X. Fu, T. Li, Aurora: a thermally resilient photonic network-on-chip architecture, in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 379–386 (2012). doi:10.1109/ICCD.2012.6378667
73. A. Rival, D. Jary, C. Delattre, Y. Fouillet, G. Castellan, A. Bellemin-Comte, X. Gidrol, An EWOD-based microfluidic chip for single-cell isolation, mRNA purification and subsequent multiplex qPCR. Lab Chip **14**(19), 3739–3749 (2014)
74. H. Rong, R. Jones, A. Liu, O. Cohen, D. Hak, A. Fang, M. Paniccia, A continuous-wave Raman silicon laser. Nature **433**, 725–728 (2005)
75. S. Roy, B.B. Bhattacharya, S. Ghoshal, K. Chakrabarty, Theory and analysis of generalized mixing and dilution of biochemical fluids using digital microfluidic biochips. ACM J. Emerg. Technol. Comput. Syst. **11**(1), Article. 2 (2014)
76. M. Saeedi, I.L. Markov, Synthesis and optimization of reversible circuits - a survey. ACM Comput. Surv. **45**, 21:1–21:34, Article 21 (2011)
77. L. Schlitt, P. Kalla, S. Blair, A methodology for thermal characterization abstraction of integrated opto-electronic layouts, in *International Conference on VLSI Design*, pp. 270–275 (2016)
78. H.-H. Shen, S.-K. Fan, C.-J. Kim, D.-J. Yao, EWOD microfluidic systems for biomedical applications. Microfluid. Nanofluid. **16**(5), 965–987 (2014)
79. V.V. Shende, A.K. Prasad, I.L. Markov, J.P. Hayes, Synthesis of reversible logic circuits. IEEE Trans. CAD **22**(6), 710–722 (2003)
80. C. Shih, Z. Zeng, C. Shiuh, Extinction ratio compensation by free carrier injection for a MOS-capacitor microring optical modulator subjected to temperature drifting, in *CLEO/PACIFIC RIM '09*, pp. 1–2 (2009). doi:10.1109/CLEOPR.2009.5292625
81. P.W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, in *Proceedings of Foundations of Computer Science*, pp. 124–134 (1994)
82. R. Sista, Z. Hua, P. Thwar, A. Sudarsan, V. Srinivasan, A. Eckhardt, M. Pollack, V. Pamula, Development of a digital microfluidic platform for point of care testing. Lab Chip **8**(12), 2091–2104 (2008)
83. M. Soeken, S. Frehse, R. Wille, R. Drechsler, RevKit: an open source toolkit for the design of reversible circuits, in *Reversible Computation 2011*. Lecture Notes in Computer Science, vol. 7165 (Springer, Berlin, 2012), pp. 64–76. RevKit is available at www.revkit.org
84. M. Soeken, R. Wille, O. Keszocze, D.M. Miller, R. Drechsler, Embedding of large Boolean functions for reversible logic. J. Emerg. Technol. Comput. Syst. **12**(4), 41:1–41:26, Article 41 (2015). https://doi.org/10.1145/2786982
85. R. Soref, The past, present, and future of silicon photonics. IEEE J. Sel. Top. Quantum Electron. **12**(6), 1678–1687 (2006). doi:10.1109/JSTQE.2006.883151
86. F. Su, K. Chakrabarty, Architectural-level synthesis of digital microfluidics-based biochips, in *Proceedings of IEEE International Conference on CAD (ICCAD)*, pp. 223–228 (2004)

87. F. Su, K. Chakrabarty, Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips, in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 825–830 (2005)
88. F. Su, K. Chakrabarty, Module placement for fault-tolerant microfluidics-based biochips. ACM Trans. Des. Autom. Electron. Syst. **11**, 682–710 (2006)
89. F. Su, K. Chakrabarty, High-level synthesis of digital microfluidic biochips. ACM J. Emerg. Technol. Comput. Syst. **3**(4), Article 16 (2008)
90. F. Su, W. Hwang, K. Chakrabarty, Droplet routing in the synthesis of digital microfluidic biochips, in *Proceedings of Design, Automation and Test in Europe (DATE)*, pp. 323–328 (2006)
91. M.K. Thomsen, A functional language for describing reversible logic, in *Forum on Specification and Design Languages*, pp. 135–142 (2012)
92. L.M.K. Vandersypen, M. Steffen, G. Breyta, C.S. Yannoni, M.H. Sherwood, I.L. Chuang, Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. Nature **414**, 883 (2001)
93. G.F. Viamontes, M. Rajagopalan, I.L. Markov, J.P. Hayes, Gate-level simulation of quantum circuits, in *ASP Design Automation Conference*, pp. 295–301 (2003)
94. R. Wille, R. Drechsler, BDD -based synthesis of reversible logic for large functions, in *Design Automation Conference*, pp. 270–275 (2009)
95. R. Wille, D. Große, G. Dueck, R. Drechsler, Reversible logic synthesis with output permutation, in *VLSI Design*, pp. 189–194 (2009)
96. R. Wille, D. Große, S. Frehse, G.W. Dueck, R. Drechsler, Debugging of Toffoli networks, in *Design, Automation and Test in Europe*, pp. 1284–1289 (2009)
97. R. Wille, D. Große, D.M. Miller, R. Drechsler, Equivalence checking of reversible circuits, in *International Symposium on Multi-Valued Logic*, pp. 324–330 (2009)
98. R. Wille, S. Offermann, R. Drechsler, SyReC: a programming language for synthesis of reversible circuits, in *Forum on Specification and Design Languages*, pp. 184–189 (2010)
99. R. Wille, D. Große, S. Frehse, G.W. Dueck, R. Drechsler, Debugging reversible circuits. Integration **44**(1), 51–61 (2011)
100. R. Wille, O. Keszöcze, R. Drechsler, Determining the minimal number of lines for large reversible circuits, in *Design, Automation and Test in Europe* (2011)
101. R. Wille, R. Drechsler, C. Osewold, A.G. Ortiz, Automatic design of low-power encoders using reversible circuit synthesis, in *Design, Automation and Test in Europe*, pp. 1036–1041 (2012)
102. R. Wille, M. Soeken, D.M. Miller, R. Drechsler, Trading off circuit lines and gate costs in the synthesis of reversible logic. Integration **47**(2), 284–294 (2014)
103. R. Wille, O. Keszocze, C. Hopfmuller, R. Drechsler, Reverse BDD-based synthesis for splitter-free optical circuits, in *Asia and South Pacific Design Automation Conference*, pp. 172–177 (2015)
104. R. Wille, O. Keszocze, S. Hillmich, M. Walter, A.G. Ortiz, Synthesis of approximate coders for on-chip interconnects using reversible logic, in *Design, Automation and Test in Europe*, pp. 1140–1143 (2016). http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=7459481
105. R. Wille, E. Schönborn, M. Soeken, R. Drechsler, SyReC: a hardware description language for the specification and synthesis of reversible circuits. Integr. VLSI J. **53**, 39–53 (2016)
106. B.S. Wheeler, J.A. Blau, H.F. Willard, K.C. Scott, The impact of local genome sequence on defining heterochromatin domains. PLoS Genet. **5**(4), 1–15 (2009)
107. B.S. Wheeler, B.T. Ruderman, H.F. Willard, K.C. Scott, Uncoupling of genomic and epigenetic signals in the maintenance and inheritance of heterochromatin domains in fission yeast. Genetics **190**(2), 549–557 (2012)
108. T. Xu, K. Chakrabarty, Integrated droplet routing in the synthesis of microfluidic biochips, in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 948–953 (2007)
109. T. Xu, K. Chakrabarty, A Cross-referencing-based droplet manipulation method for high-throughput and pin-constrained digital microfluidic arrays, in *Proceedings of IEEE/ACM Design, Automation and Test in Europe (DATE)*, pp. 552–557 (2007)

110. T. Xu, K. Chakrabarty, Broadcast electrode-addressing for pin-constrained multi-functional digital microfluidic biochips, in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 173–178 (2008)
111. T. Xu, K. Chakrabarty, Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. ACM J. Emerg. Technol. Comput. Syst. **4**(3), Article 11 (2008)
112. T. Xu, K. Chakrabarty, A droplet-manipulation method for achieving high-throughput in cross-referencing-based digital microfluidic biochips. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(11), 1905–1917 (2008)
113. P.-H. Yuh, C.-L. Yang, Y.-W. Chang, BioRoute: a network flow based routing algorithm for digital microfluidic biochips, in *Proceedings of IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 752–757 (2007)
114. P.-H. Yuh, C.-L. Yang, C.-W. Chang, Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation. ACM J. Emerg. Technol. Comput. Syst. **3**(3), 13.1–13.32 (2007)
115. P. Zeitzoff, J. Chung, A perspective from the 2003 ITRS. IEEE Circuits Syst. Mag. **21**, 4–15 (2005)
116. Y. Zhao, K. Chakrabarty, Cross-contamination avoidance for droplet routing in digital microfluidic biochips, in *Proceedings of IEEE/ACM Design, Automation and Test in Europe (DATE)*, pp. 1290–1295 (2009)
117. Y. Zhao, K. Chakrabarty, Synchronization of washing operations with droplet routing for cross-contamination avoidance in digital microfluidic biochips, in *Proceedings of IEEE/ACM Design Automation Conference (DAC)*, pp. 635–640 (2010)
118. A. Zulehner, R. Wille, Make it reversible: efficient embedding of non-reversible functions, in *Design, Automation and Test in Europe* (2017)
119. A. Zulehner, R. Wille, Taking one-to-one mappings for granted: advanced logic design of encoder circuits, in *Design, Automation and Test in Europe* (2017)

Chapter 4
Physical Awareness Starting at Technology-Independent Logic Synthesis

André Inácio Reis and Jody M.A. Matos

1 Introduction

Designing digital circuits is an extremely complex task, specially for advanced technology nodes. This way, electronic design automation [14, 38, 43] tools are extensively used to complete the design. Even with this extensive use of computer-aided design tools, the path from an initial description to a layout ready to tape-out is a long and laborious process, involving several different steps performed by specialized teams. The sequence of steps performed to produce the final circuit is called design flow.

Different design flows are possible, in which nearly all of them present small variations from each other. Most of these variations rely on the type of design being synthesized and/or on changes from one company to another. However, there are some steps that may be considered standard. Early steps include logic synthesis [11, 15, 16, 20, 31], where an initial circuit description is implemented as a set of interconnected logic gates, possibly from a cell library [30]. Later steps include physical design [3, 18, 23, 34], where the set of interconnected logic gates is placed and routed to form the final layout.

Historically, logic synthesis and physical design tools have been produced by different teams, or even different companies. More than that, these teams come from a different culture, having separate specialized training on either logic synthesis or physical design. Even today, there are very few people that attend the leading conferences in both areas: IWLS (International Workshop on Logic Synthesis) and

A.I. Reis (✉)
PPGC/PGMICRO, Institute of Informatics, UFRGS, Porto Alegre, RS, Brazil
e-mail: andre.reis@inf.ufrgs.br

J.M.A. Matos
PPGC, Institute of Informatics, UFRGS, Porto Alegre, RS, Brazil

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_4

ISPD (International Symposium on Physical Design). So, there is a gap between the logic synthesis and physical design cultures.

Efficient design flows should provide a tight integration between logic synthesis and physical design. However, due to the culture gap, this integration is not satisfactory. Early steps of logic synthesis work as a technology-independent step, meaning that no technology information is taken into account. Later steps of logic synthesis, known as technology-dependent optimizations, take into account information from a physical cell library. Yet, no information about placement and routing is taken into account. When physical design places and routes the cells chosen during logic synthesis, large capacitances may be introduced, fundamentally changing the assumptions that were made in the previous steps. This way, the whole process of logic synthesis and physical design must be reiterated, hoping for a convergence that may not arise.

In this chapter we discuss how to bring technology information into early steps of logic synthesis. For this, we rethink the design flow as a whole, redefining steps and their interactions. We propose to achieve a better integration through a paradigm shift, where the flow is divided into signal distribution and signal computation. We discuss how to implement this paradigm shift with a set of enablers. As a consequence, both signal distribution and signal computation can benefit from tasks pertaining to logic synthesis and physical design. This way, the emphasis is shifted from what is logic synthesis and what is physical design. The focus is shifted to performing logic synthesis and physical design of signal distribution circuits before performing logic synthesis and physical design of logic computation circuits. The proposed paradigm shift allows the power of logic restructuring using Boolean methods to be unleashed without the fear that long wires introduced later during physical design would invalidate assumptions made during logic synthesis.

This chapter is organized as follows. Section 2 presents some preliminary concepts related to the performance of digital integrated circuits. Section 3 describes a typical design flow. Section 4 discusses common design tasks in a typical design flow. Tasks are described to point out what information is needed to perform each task, determining when they can be performed during the flow. Section 5 presents some features that act as enablers to bring technology information into early steps of the design flow. Section 6 proposes a new physically aware design flow focused on designing signal distribution circuits prior to the design of logic computation circuits. This way tasks which bring physical information important for circuit performance can occur earlier in the design flow. Finally, Sect. 7 concludes the chapter.

2 Basic Concepts in VLSI Design

In this section, we present some basic concepts related to VLSI design. The emphasis is on the role played by these concepts in the design flow, specially concerning circuit performance.

2.1 Design Constraints

Design constraints are used by VLSI engineers to express design intent. Typically, VLSI engineers use a design constraints file to express design goals, such as the desired working frequency, maximum area, and maximum power consumption for a specific design.

For instance, design constraints can be used to direct the tools that compose a VLSI design flow to minimize power consumption while satisfying a specific target frequency requirement. This way, design tools must be able to understand and cope with design intent expressed through design constraints. Preferentially, the treatment of design constraints should start at early steps in the design flow.

2.2 Sources of Delay

Target design performance is an extremely important constraint for VLSI circuits. This aspect is mainly related with the delay on critical paths compared to the period associated with the required frequency. This way, it is key to understand that delays may come from different sources. This idea is illustrated in Fig. 4.1. Every path in a circuit has a delay, and part of this delay come from three different sources: (1) delay from late arrival (or early required) times; (2) cell (gate) delays; and (3) wire delays due to signal distribution (wires and buffers). For instance, in Fig. 4.1, the required time for the whole circuit is larger than the current circuit delay. In this case, either the wire delay or the gate delay should be optimized (reduced) to achieve the required time. Generally speaking, late arrivals are part of the problem definition and cannot be optimized. We will discuss these types of delays in the next subsections.

2.2.1 Delay from Late Arrival

Some signals may arrive late, or may be required to be delivered earlier with respect to the required time. These conditions are typically described in the design constraints file, that specifies the boundary conditions for the circuit. In fact, these

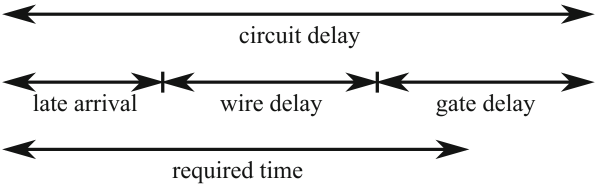


Fig. 4.1 Different sources of delay in a circuit

delays are simply given in the constraints file and the designer must cope with them by reducing gate and wire delay to accommodate non-negotiable late arrival times, such that the final required time for the circuit is achieved. As we highlighted before, late arrivals are part of the problem definition and cannot be optimized.

2.2.2 Cell Delay Estimation

A widely used and well-established approach to compute circuit delay is based on the Non-Linear Delay Model (NLDM). In this section, we try to briefly describe the characteristics of the NLDM delay model.

According to the NLDM delay model, the overall logic-stage delay is affected by (1) the slope of the input signal transition and (2) the lumped capacitance at the output. In practical use, the NLDM approach models the cell delay by pre-characterizing it as a matrix of delays indexed by input slope and output capacitance. The output slope is also pre-computed and stored in a similar matrix (indexed by input slopes and output capacitances). For index values that are not directly available in the matrix, approximations are obtained by linear interpolation or extrapolation.

2.2.3 Wire Delay Estimation

Once the cell delay and output transition time are efficiently evaluated, the wire delay can be calculated by driving its RC circuit model with the voltage waveform defined by a transition time. The overall stage delay is, then, the cell delay obtained from the look-up tables added to interconnect delay obtained from its RC model.

It is important to remark that, in recent technologies, intra-cell resistances and wire routing resistances can interact if they are of the same order of magnitude. As a consequence, part of the lumped output capacitance may be hidden from the driver cell, leading to the need of computing an effective capacitance. When a source drives multiple sinks, the effective capacitance for each sink may be different. We refer the reader to the work in [36] for details.

2.3 Timing Closure

VLSI designs must meet the required timing constraints, e.g. setup (long-path) and hold (short-path) constraints. The optimization process that guarantees these requirements is called timing closure. This process integrates locally performed optimizations, including modifications of placement and routing, with methods oriented to improve circuit performance.

However, meeting the required timing constraints for a design is not the only goal when running for timing closure anymore. The naive usage of timing-driven approaches could lead to undesired scenarios. For instance, consider a design that meets the timing constraints, but where a number of non-critical paths were over optimized, resulting in unnecessary area and power payloads.

This way, clever timing closure procedures try to find the compromise of zero-slack designs. In such designs, the circuit has fairly optimized critical paths with no unnecessary payloads. At the same time, non-critical paths are cleverly relaxed, in order to avoid undesired area and power overheads.

2.4 Timing Budget

In timing-driven physical design, both cell and wire delays must be optimized to obtain a timing-correct layout. However, there is a cause-and-effect dilemma: (1) timing optimization requires knowledge of capacitive loads and, hence, actual wirelength, but (2) wirelengths are unknown until placement and routing are completed. To help resolve this dilemma, timing budgets are used to establish delay and wirelength constraints for each net, thereby guiding placement and routing to a timing-correct result. A popular approach to timing budgeting is the zero-slack algorithm (ZSA), and we refer the reader to [33] for details. From our standpoint, timing budget can be viewed as a partitioning of the complete circuit design constraints into sub-design constraints for sub-circuits and associated interconnections.

2.5 Design Convergence and Delay Information Stability

Wire delay and cell delay must be optimized during the design cycle. In order to achieve design convergence, these delays have to be estimated in a precise and stable way. By stable, it is understood that assumptions will not be changed so that delay information is changed. It is very difficult to estimate wire lengths early in the design flow. This way, instability and lack of design convergence arise not because the sizes of cells were not adequately chosen, but instead because routing capacitances from long wires can invalidate assumptions made at the early steps of the design flow.

3 A Typical Flow

A typical design flow roughly follows the steps presented in Fig. 4.2. Notice that the flow is separated into two distinct groups of tasks: Logic Synthesis (a.k.a. frontend) and Physical Design (a.k.a. backend).

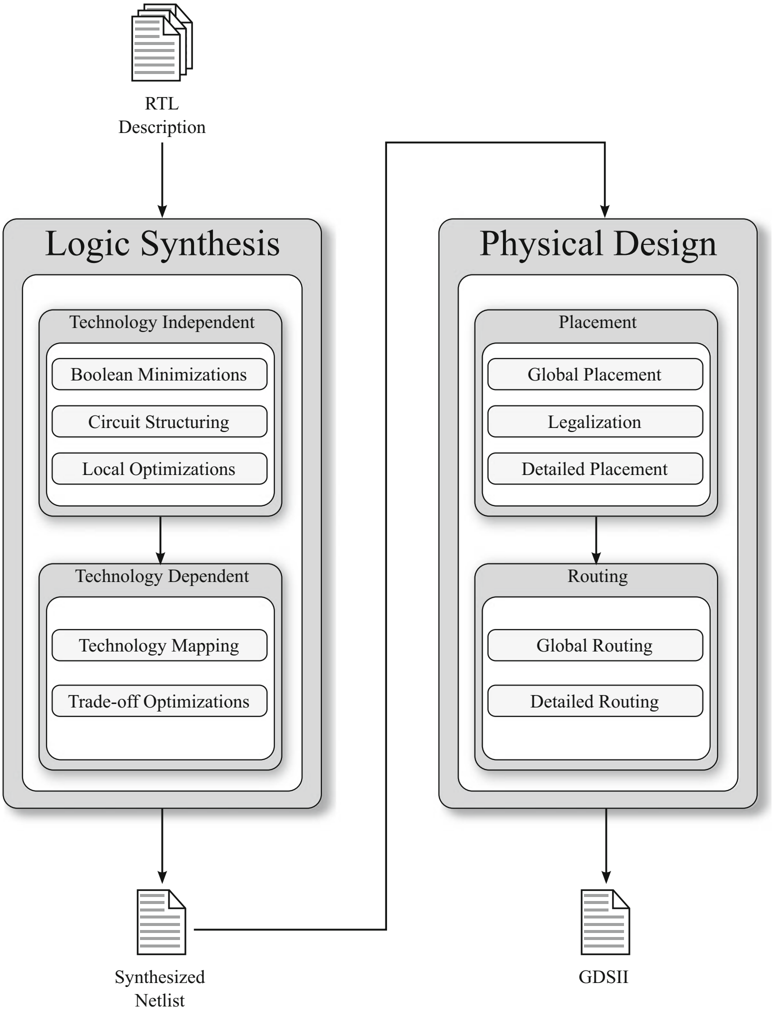


Fig. 4.2 A typical VLSI design flow

3.1 Logic Synthesis Frontend

Given an input design described in register-transfer level (RTL), a usual design flow relies on synthesis tasks commonly performed in sequence, typically seeking for timing closure. Thus, the RTL description is initially transformed into a technology-independent logic representation and a first level of logic synthesis and associated optimizations are performed. Then, such a logic representation is mapped into instances of a cell library and a second level of logic synthesis and optimizations are performed, now under a technology-dependent perspective. This process is responsible for generating a mapped netlist. It also defines a sort of checkpoint on the design closure pace, in the sense that both frontend and backend designers try to agree that the design closure is achievable given the generated netlist.

3.2 Physical Design Backend

The mapped netlist is, then, taken as input for physical design. The first step is mainly based on defining the die area and placing both the I/O pins and macro-blocks. As soon as the design is floorplanned, steps so-called global placement, legalization and detailed placement are responsible for placing the cells from the netlist onto the chip's surface while optimizing multiobjective cost functions. Then, the interconnection net topologies are defined and their segments are assigned into appropriate routing layers through global and detailed routing.

3.3 Drawbacks

Although the flow is somehow convergent, achieving such a convergence has been even more challenging at each new technology node. Also, each of these synthesis steps is based on incremental optimizations, which sometimes need to be repeatedly iterated before moving to the following steps. Still, it is increasingly common that the flow does not converge in the end and demand for multiple iterations over the entire design flow before to achieve design closure. Thus, it is desirable to look for techniques to achieve the desired timing constraints spending less design cycles, i.e., minimizing the number of iterations in the design flow.

We believe that considering physical information since the early steps of logic synthesis can potentially minimize the number of design cycles. However, the way current flows are organized limits how physical information can be incorporated in the design flow. In the next sections, we describe common synthesis tasks and the moment when they can be performed during a typical design flow.

4 Common Synthesis Tasks

In this section, we present some synthesis tasks commonly performed on VLSI design flows. This list of tasks was originally presented by Alpert [1] as a small part of his IWLS 2013 invited talk, entitled “When Logic Synthesis Met Physical Synthesis.” The goal during the keynote was to discuss with the audience if these tasks belonged into the domains of logic synthesis or physical design. Herein, we discuss what type of information is needed to perform each of these tasks and how this limits the points where they can be performed during a typical design flow.

4.1 Gate Sizing or Repowering

The idea of gate sizing, or repowering, is that each logic cell in a library is provided with several options of drive strengths. For instance, a certain library could have several versions of a 2-input NAND cell. These versions would have different drive strengths, such as NAND2_X1, NAND2_X2, NAND2_X4, and NAND2_X8. These cells implement the same logic function, but internal transistors are sized differently so that the cells will have different input capacitances, different physical sizes, and different output drive strengths.

The goal of gate sizing is to reduce total circuit area or power while respecting the maximum delays expressed through the user-defined design constraints. The solution to the problem of gate sizing chooses a specific size for each cell with the goal of minimizing the adopted cost function while still respecting the timing constraints. Figure 4.3 illustrates a cell being gate-sized in a given circuit.

To be meaningful, the gate sizing procedure has to have a notion of the capacitances to be driven by each cell. This is necessary because the solution to the problem has to adjust the drive strength of each cell to the output capacitance that the cell has to drive.

Gate sizing can be performed after technology mapping, during the technology-dependent phase of logic synthesis. However, at this point of the design flow, output wire capacitances are still estimated and can change after place and route. This way, gate sizing may be performed again after place and route when extracted wire capacitances are known. One problem with this flow is that the sizes of some cells may increase in such a way that there is not sufficient room for them in the

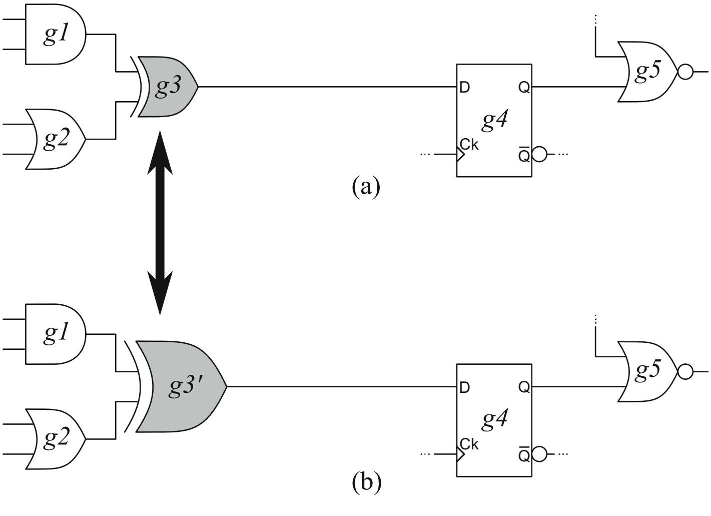


Fig. 4.3 An example of gate sizing or repowering: the instance g3 has a smaller cell size in option (**a**), compared to instance g3′ in option (**b**)

current placement, and consequently the process has to be repeated until it hopefully converges, which is not guaranteed. Several authors published works on gate sizing [4, 9, 17, 19].

4.2 Vt Swapping

The same way a cell is provided with different drive strengths, each cell instance can have different threshold voltages (Vt) assigned to its transistors. A high-Vt cell is slower, but consumes less leakage power. In contrast, a low-Vt cell is faster, but consumes more leakage power. Swapping the Vt of individual cells allows to trade timing slacks for leakage-power reduction.

The goal of Vt swapping is to reduce leakage power while respecting the target delays expressed through the user-defined design constraints. The solution to the problem of Vt swapping chooses a specific Vt for each cell with the goal of minimizing the leakage power while still respecting the timing constraints. Figure 4.4 depicts a circuit before (option (a)) and after (option (b)) Vt swapping.

To be meaningful, the Vt swapping also needs to have a notion of both the capacitances to be driven by each cell and the input transition slope, just like for gate sizing. This is necessary because the solution to the problem has to adjust the cell Vt while respecting timing constraints, and this is the information commonly required for timing estimations.

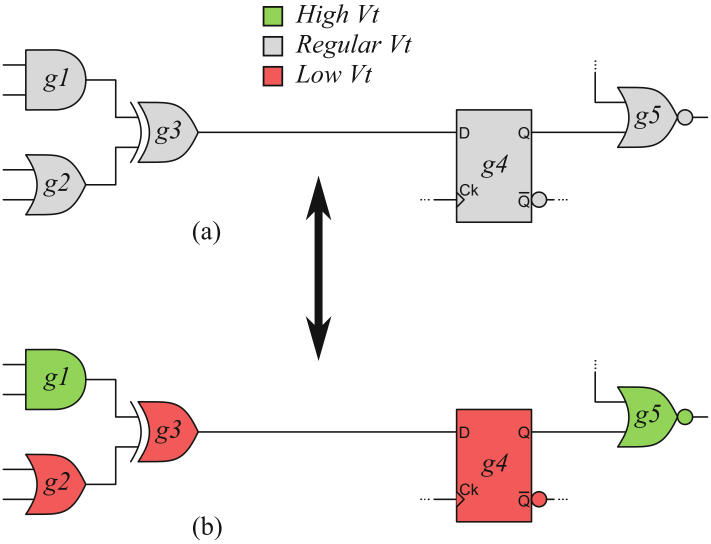


Fig. 4.4 An example of vt-swapping: in option (**a**) all instances have regular Vt, in option (**b**) some cells have high Vt and others have low Vt

Vt swapping can be performed after technology mapping, during the technology-dependent phase of logic synthesis. However, at this point of the design flow, output wire capacitances are still estimated and can change after place and route. Thus, this task has also to be performed after place and route when extracted wire capacitances are known. Works that address the task of Vt swapping include [24, 40].

4.3 Cell Movement

A major problem in the VLSI design flow relies on the challenge of finding good locations for individual circuit components. The task is, then, arranging circuit components on the die surface while optimizing multiobjective cost functions, which are commonly timing-driven or routability-aware.

The intuition for the cell movement/placement task is based on modeling the circuit as a graph, in which the cells are the nodes and the interconnections are the edges. The problem is, therefore, defined by the assignment of (x, y) locations for each of those cells in such a way that the adopted cost functions are minimized and the design constraints are met. Figure 4.5 provides an example of cell movement.

Placers usually need to have a notion of the die area, the I/O pin placement and both the standard height and width step of the cells (a.k.a. placing sites). Along with both a netlist description, the logic library and the physical library, this information can be used to formally model the problem of placing cells into a discrete matrix.

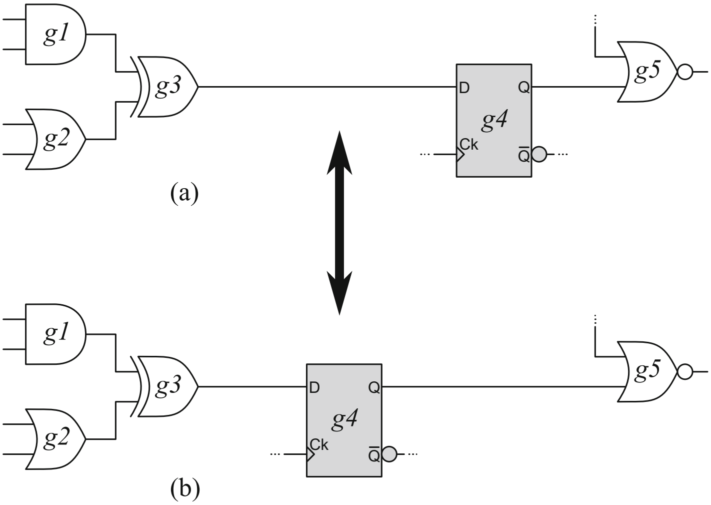


Fig. 4.5 An example of cell-movement: the position of instance g4 is different in options (**a**) and (**b**), this changes wire lengths and associated parasitics, impacting the timing

In a standard design flow, the first placement iteration is ran after logic synthesis, as soon as the netlist is defined, and the floorplanning is performed. Further in the flow, cells can change location in almost all of the subsequent steps, including local optimization steps. Authors that use cell movement as part of their optimization procedures include [21, 27].

4.4 Layer Assignment

Another important task in the VLSI design flow is the layer assignment for routing the circuit interconnections. Considering that VLSI technologies allow for multiple routing layers, with different electrical characteristics, layer assignment is the task of properly selecting which routing layer should be used when defining the path of each net segment. Higher routing layers are thicker and both less resistive and less capacitive, but are less available as routing resources. In contrast, lower routing layers offer a high number of routing resources, but they are thinner and both more resistive and more capacitive.

Taking the current placement solution and a routing topology for a given net, the problem relies on assigning a routing layer for each segment of this topology. Figure 4.6 illustrates an example of this task. The assignment must consider: (1) the routing congestion; (2) the available routing resources for the different routing layers; and (3) the layers' electrical characteristics. At the end, the proposed

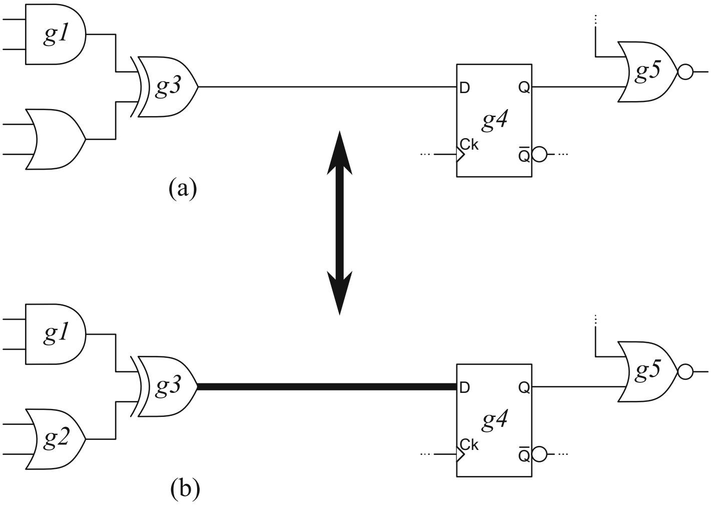


Fig. 4.6 An example of layer-assignment: the wire connecting instances g3 and g4 is assigned to a different layer in options (**a**) and (**b**), this changes wire layer properties and associated parasitics, impacting the timing

solution needs to satisfy the fabrication process design rules, complete all the circuit connections, and meet their required timing budgets while optimizing a variety of cost functions, such as wirelength-related or lithography-aware optimizations.

Layer assignments are primarily performed as part of the routing stage, which, in turn, occurs once the circuit is placed. Also, this task takes its place on almost every post-routing optimization run. We refer the reader to the following works as example of layer assignment methods [6, 10].

4.5 Buffering Long Nets

Since the interconnect delay is progressively playing an even more dominant role for the systems performance, buffer planning became a critical step of the design flow. Considering that long nets would have higher capacitance (and, consequently, lower performance), buffering long nets is a task to pay attention.

The challenge is, then, to decide whether buffers should be inserted (or deleted) in order to improve the overall design performance while tracking both placement and routing congestion. Figure 4.7 depicts a buffering scenario.

Assuming that this task needs to evaluate wirelength to have a notion of long nets, buffering for interconnect delay optimization is usually ran after placement. Thus, along with the circuit representation and the cell positions, this buffering process needs the standard cell library and at least a rough information about the wires' electrical characteristics. Buffering of long nets is addressed in [2, 42].

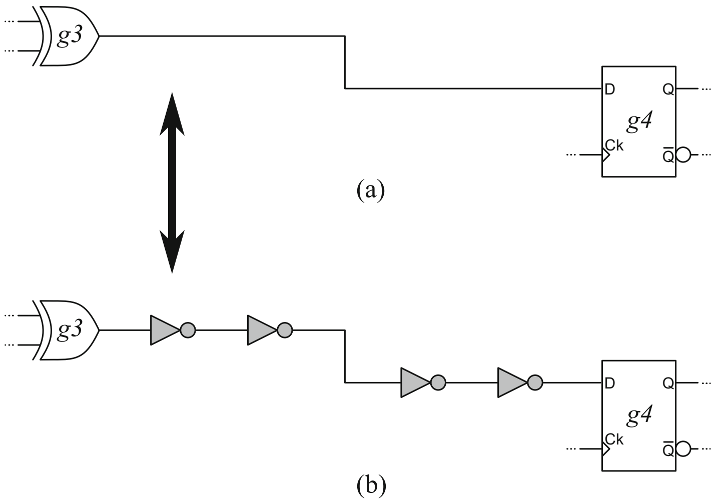


Fig. 4.7 An example of buffering long nets: the wire between instances g3 and g4 has no buffer in option (**a**) and four buffers inserted in option (**b**)

4.6 Buffer Deletion

The previous section discussed buffering of long nets. Part of this task may include buffer deletion, as depicted in Fig. 4.8. Works that perform buffer deletion as part of the optimization include [5, 32].

4.7 Buffering Nets to Reduce Fanout

Considering that the cell delay component is highly dependent on its output load capacitance, reducing the cell fanout is an important optimization task. Since the higher is a cell fanout, the higher tends to be its output load capacitance, buffers and inverters can be used to reduce the cell fanout looking for delay optimizations. Figure 4.9 illustrates a fanout optimization by buffer insertion. We refer the reader to the works [5, 32], as examples of buffering nets to reduce fanout.

4.8 Pin Swapping

When two cell pins are functionally symmetric, the wires connected to them can be swapped without changing the overall circuit functionality. This is the simplest form of applying rewiring algorithms, a.k.a. pin swapping approaches.

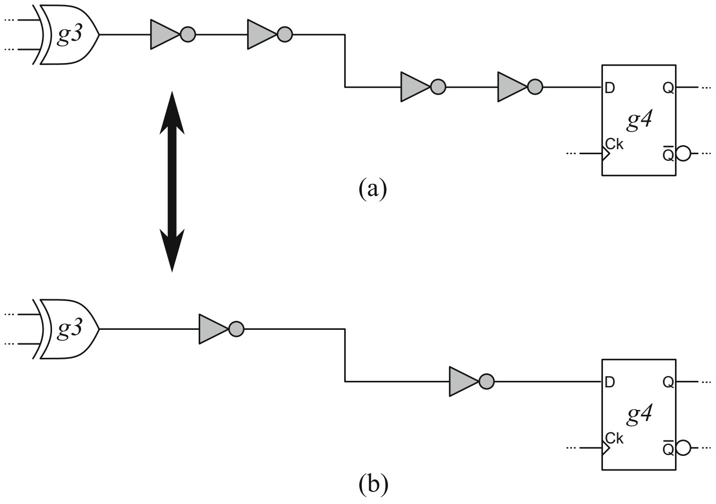


Fig. 4.8 An example of buffer deletion: the wire between instances g3 and g4 has four buffers in option (**a**) and just two buffers in option (**b**)

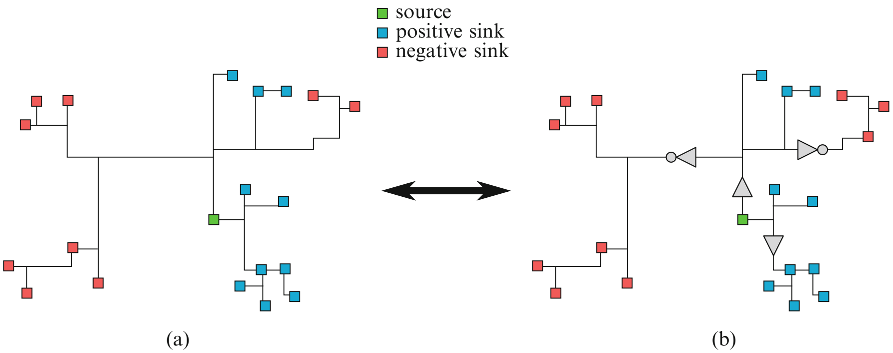


Fig. 4.9 An example of buffering to reduce fanout: option (**a**) was not yet buffered and has an excessive fanout, with nonadjusted polarities; option (**b**) has three buffers inserted to reduce fanout and also to adjust the polarity of sinks

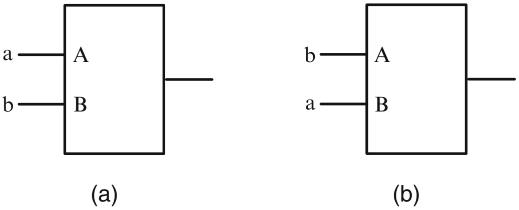


Fig. 4.10 An example of pin swapping: (**a**) before swapping and (**b**) after swapping, with input signals interchanged between symmetric inputs

Rewiring techniques are commonly used to reduce circuit delay or power consumption, since rewiring allows to reduce routing wirelength and it can also minimize routing congestion. Figure 4.10 depicts a rewiring scenario.

Pin swapping techniques aim the detection of circuit symmetries, which could yield delay or power optimizations, while keeping the design functionality. This is usually achieved by symmetry techniques on top of either structural representations of a given circuit (such as an AIG or a netlist) or Boolean representations (such as a binary decision diagram). Beyond swapping symmetric inputs of a cell, state-of-the-art algorithms allow for simultaneous permutations of multi-variable symmetric inputs and outputs of subcircuits.

When targeting cell-oriented pin swapping, the symmetry for input pins can be obtained from the physical library. Symmetry detection algorithms for subcircuits typically extend this concept for inputs and outputs permutations. In both of the cases, state-of-the-art approaches are ran after placement, in order to efficiently evaluate wirelength reductions. Nevertheless, there exist rewiring algorithms with different cost functions, which can be applied since the early steps of logic synthesis. Works that perform pin swapping as part of the optimization strategy include [7, 8].

4.9 Cloning

Gate duplication, cell replication, or simply cloning, is the process of duplicating a gate in a given circuit in order to reduce fanout. Once the fanout is reduced, both gates (the original and the cloned) tend to have better performance than before replication.

State-of-the-art algorithms for gate cloning rely on traversing the circuit a few times by looking for potentially “clonable” cells, choosing the ones to be cloned, then performing the cloning itself. Figure 4.11 illustrates the cloning task.

Typical inputs for a cell replication algorithm are a graph representation of the circuit and delay analysis for timing-driven approaches. Cloning is a task usually ran during logic synthesis, but not rarely it is also applied for post-placement optimizations. We refer the reader to the works in [22, 35, 41] as examples of methods that perform cloning.

4.10 Balancing and Unbalancing of AND/OR/XOR Trees

Digital circuits may have large operator trees, which perform a large AND/OR/XOR operation. These trees consist of several interconnected gates of the same type, forming a large operator.

Large trees provide a low effort opportunity to optimize a circuit, specially concerning timing optimizations [44], as the tree can be reordered to alter the depth of the tree. This can be used to accommodate different arrival times as well as to facilitate routing after placement.

Tree balancing can be performed during the logic synthesis steps to reduce logic depth, as illustrated in Fig. 4.12. It can also be performed after physical design considering placement and wire delay, as shown in Fig. 4.13.

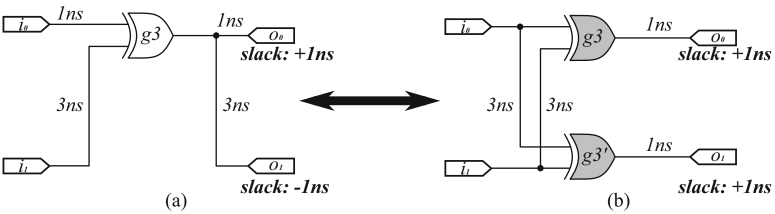


Fig. 4.11 An example of cloning: the instance g3 in option (**a**) has been cloned into two instances g3 and g3′ in option (**b**), allowing to recover negative slacks

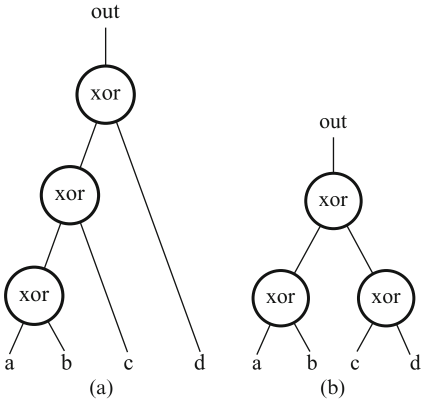


Fig. 4.12 Balancing a XOR logic tree during logic synthesis to reduce logic depth. Option (**a**) has three logic levels, but it can be useful if input d is the only signal in the circuit critical path. Option (**b**) has two logic levels and it is a balanced structure, as all four inputs have a logic level depth of two

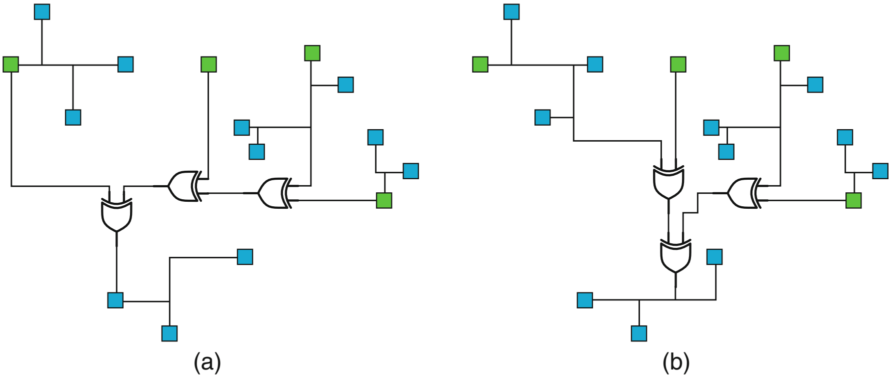


Fig. 4.13 Balancing a XOR logic tree after place and route to reduce logic depth and improve routing. Option (**a**) is logically equivalent to Fig. 4.12a. Option (**b**) is logically equivalent to Fig. 4.12b. Notice that the rebalancing can have positive impact on circuit wiring if done after place and route

4.11 Composition/Decomposition

A variety of trade-off optimizations also need to tackle the composition/decomposition dilemma. While playing with the circuit granularity, different cost functions are optimized according to where in the design flow the optimizations are taken. Common cost functions range from literal counting to the usual area, power, and timing (or levels). Figure 4.14 illustrates the process, in which the option (a) can be thought as decomposed from option (b), or, conversely, option (b) can be viewed as a composition from option (a).

Logic decomposition is the process of breaking down complex functions into simpler components. On the other way around, logic composition is the process of building up complex functions out of simpler components. Since that the quality of results of a technology mapping depends significantly on the initial description of the circuit, the final implementation varies reasonably according to the adopted (de)composition strategy.

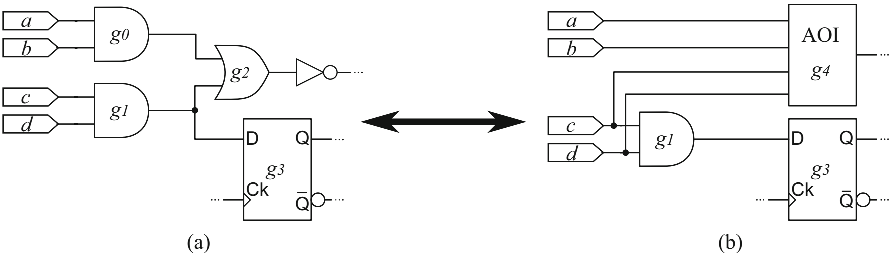


Fig. 4.14 An example of composition: instances g0 and g2 in option (**a**) were composed into a single instance g4 in option (**b**)

Both composition and decomposition approaches can be ran in almost any stage of the design flow, since technology-independent logic synthesis to post-routing optimizations. However, the later is the stage in the design flow, the lower is the degree of freedom for (de)composition-based techniques.

Apart from that, whatever is the stage in the design flow, the typical inputs for composition/decomposition algorithms are the same ones for technology mapping. Thus, along with a circuit representation, it is also commonly needed a description of the timing constraints and a standard cell library. This information would be used for timing evaluations and help the decomposition process to take the necessary decisions [39].

4.12 MUX Decomposition

The MUX decomposition is a special case of the general (de)composition. Typically, decomposition schemes target two-input gates and inverters, which excludes in the process certain important circuit elements, such as multiplexers. In the example depicted in Fig. 4.15, the AOI solution (option (a)) has $10 + 10n$ literals and the max fanout is n. The AND2 solution (not shown) has $10 + 12n$ literals and the max fanout is also n. The MUX solution (option (b)) has $9n$ literals and the max fanout is $2n$ [39].

4.13 Inverter Absorption (Decomposition)

During technology mapping, the algorithms must also consider matching solutions in both polarities, which may include decisions related to either making inverters explicit or “absorbing” them into the mapped cell. Figure 4.16 illustrates an inverter absorption possibility. In this case, the tool is able to choose between making the inverter explicit (option (a)) or absorbing it into the XOR2 cell, turning it into a XNOR2 cell. These decisions are commonly associated to area minimization and

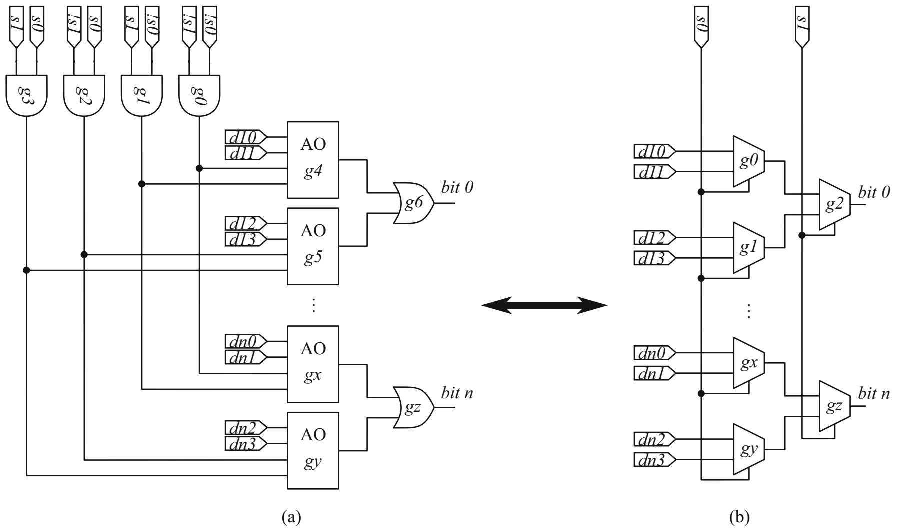
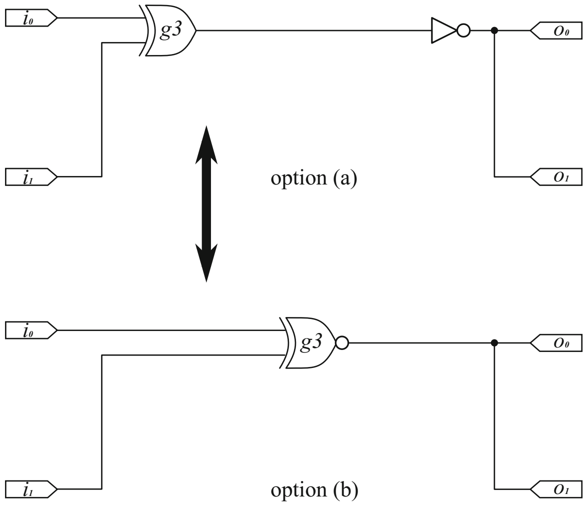


Fig. 4.15 An example of mux decomposition: complex gates in option (**a**) were decomposed into 2to1 MUXes in option (**b**)

Fig. 4.16 An example of inverter absorption: the inverter in option (**a**) was absorbed by gate g3 in option (**b**)



fanout reduction. In order to increase the number of possible matches, seeking for better quality of results, it is possible to decompose the circuit representation by making inverters explicit even when the signal should not be inverted at all. This would allow the mapper to decide whether the inverters should be absorbed or not.

Given a circuit representation, the decomposition for inverter absorption relies on decomposing such a representation in order to allow for optimizations while absorbing inverters into cells (or even making them explicit). A well-established algorithm for such a decomposition propose to (1) replace every wire in the circuit by a pair of inverters and (2) adding a “wire” cell to the library (whose pattern consists of a pair of inverters connected in series). A covering algorithm can be ran in order to seek for covering solutions that minimizes the desired cost function.

Decompositions for inverter absorption usually take place during logic synthesis. Nevertheless, inverters can also be absorbed (or turned explicit) in any post-placement or post-routing optimization. To be meaningful, this task must also have the same inputs of that for technology mapping [37].

4.14 Potential for Improvement

We believe that the availability of stable information about long wire delays since early steps of logic synthesis could potentially allow the tasks described in this section to become part of technology dependent logic synthesis, minimizing the number of design cycles. The following section presents the enablers for the wire physical awareness flow proposed in this work.

5 Enablers of Physical Awareness Flow

In this section, we discuss some features that are very important as enablers of physical awareness at early design steps. Four different enablers of physical awareness and their roles in the design flow are discussed in the following subsections.

5.1 PAIGs

The first proposed feature to enable physical awareness are placed and-inverter graphs (PAIGs). Early steps of logic synthesis, start by abstracting a subject description that typically is an And-Inverter Graph (AIG) or a similar data structure. Other examples of this type of data structure include majority inverter graphs (MIGs) and xor-and-inverter graphs (XAIGs).

If we concentrate in the case of AIGs, they are very near to an implementation with simple gates (and-nand-or-nor) netlist. In order to consider physical design information during logic synthesis, it is possible to store an (x, y) position for each AIG node. Although this is an important enabler to consider physical design, it has two weaknesses. The first one is that AIGs are too fine-grained for placement. The second-one is that AIGs do not have explicit inverters, and inverters/buffers are important for signal distribution, specially for long wires.

5.2 KL-Cut PAIGs

In order to have a coarse-grained data structure for placement, it is possible to use the concept of KL-cuts [25, 26, 28]. KL-cuts are a type of cuts with at most K-inputs and L-outputs and that fanout to the neighborhood only through the L-outputs. In this sense, the placement of AIG nodes can be performed based on the KL-cut clusters, even disregarding the positions of individual nodes. The parasitics of interconnections among nodes inside KL-cuts can also be disregarded, as they are mainly local interconnections. In this sense, KL-cuts can be seen as a movable container of AIG nodes. This new data structure is a KL-cut PAIG, where the KL-cuts have an area that are proportional to the number of AIG nodes contained inside it.

5.3 Explicit Inverters on KL-Cut PAIGs

Signal distribution is done through inverters and buffers. In this sense it is important to have explicit inverters, specially for long connections. In the case of a KL-cut PAIG data structure, the explicit inverters are used in between KL-cuts. This will be explained in further detail in the next section.

5.4 Different Cuts for Signal Distribution and Logic Computation

KL-cut PAIGs are, then, formed by two main distinct cut types. The first type of cuts are the logic computation KL-cuts that act as AIG node containers, as illustrated in Fig. 4.17. The second type of cuts are signal distribution 1L-cuts (i.e., a KL-cut with K=1), which are implemented as inverter/buffer trees that have the function of distributing signals, as illustrated in Fig. 4.18. As these types of elements perform different tasks in the final circuits, they are not treated as equals. KL-cuts as node containers are local gates that compute logic, and they have to stay together in the final layout. Signal distribution trees have the function to deliver signals to distant consumer points from the local KL-cut that generates the signal. The gates that compose these inverter trees do not have to be placed together to minimize connections, but they have to be placed in order to maximize efficiency in signal distribution. The number of inverters and the internal structure of the distribution tree can vary according to the number of consumer KL-cuts, the position of consumer KL-cuts, and the required arrival times.

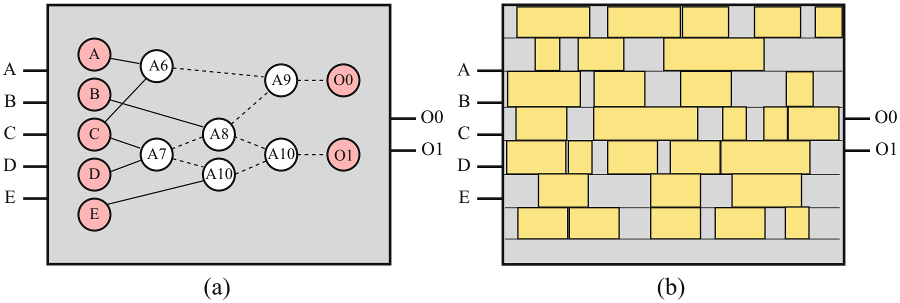


Fig. 4.17 A logic computation cut. (**a**) From a logic point of view, it is a 5-2-cut (i.e., K=5, L=2). (**b**) From a physical point of view, it is local netlist that has to be placed and routed inside predefined local boundaries

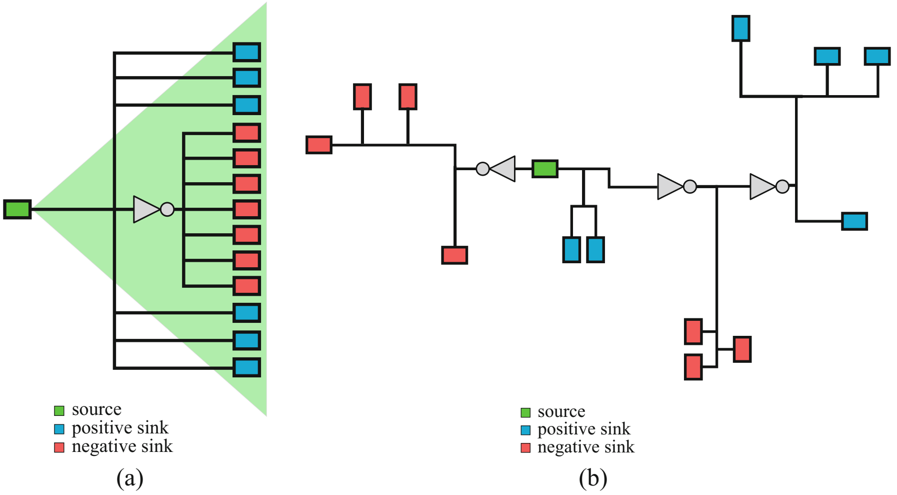


Fig. 4.18 A global signal distribution cut. (**a**) From a logic point of view it is a 1-13-cut (i.e., K=1, L=13). (**b**) From a physical point of view, it is a tree of inverters/buffers that has to be placed and routed to distribute signals according to the slacks and positions of sinks

5.5 *Local SDCs*

The last enabler of our flow is the use of local design constraints. The global design constraints are partitioned into partial design constraints for the two types of cuts (local logic computation KL-cuts and global signal distribution 1L-cuts), such that the global SDC is respected. During synthesis, the partial SDCs can be refined according to the slacks after synthesis of each signal distribution and logic computation cut.

6 Rethinking Physically Aware Flows

This section discusses how to use the enablers from previous section to produce a physically aware design flow. This flow is different from previous ones in the sense that it incorporates physical information earlier in the flow.

6.1 Starting Point and Input

The starting point of a design flow is an RTL description. In order to be processed, the initial description is abstracted to an internal data structure containing Flip-Flops and the combinational logic in between. One possibility for this data structure is a sequential AIG. In additionally to the circuit itself, the designer must specify the desired frequency through design constraints and optionally provide a floorplanning specifying position of I/O pins and available circuit area. This is illustrated in Fig. 4.19.

6.2 Placement of Interface Pins

Placement of interface pins is important for performing physically aware design, as this determines the physical position of the sources and sinks for signals. This information can be made available from a user provided initial floorplanning, or it can be computed by the tool if not specified. The point here is that the position of interface pins as illustrated in Fig. 4.19b is an important information from the start of a physically aware flow.

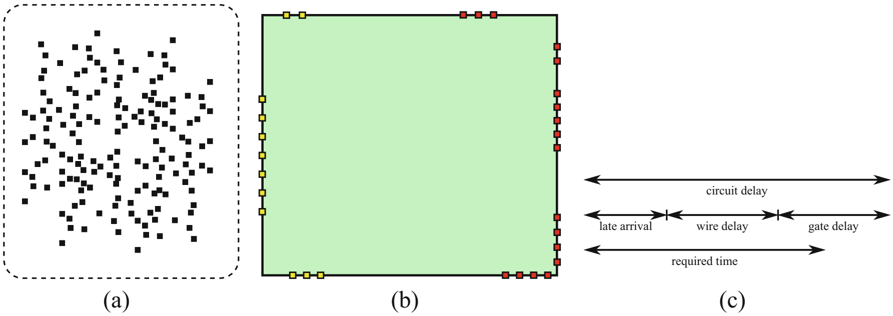


Fig. 4.19 Input data for physically aware design flow: (**a**) circuit description, (**b**) floorplan, and (**c**) design constraints

6.3 KL-Cut Computation

As we discussed before, the initial abstraction of the circuit treated by the flow is a sequential AIG, which is a description widely used in logic synthesis. As the AIG nodes will not be placed individually, but clustered into KL-cuts, a KL-cut computation is done in order to produce a KL-cut AIG to be placed. After the KL-cut computation, the available information for the flow is as depicted in Fig. 4.20. The circuit will be partitioned into KL-cuts, which can be seen as clusters of AIG nodes. The connections among the clusters and among the I/O pins are also known. However, the position of the KL-cuts is not yet final. This is represented in Fig. 4.20 as the bad quality routing with long wires. In fact, no position is taken into account while performing the KL-cuts on the initial AIG. However, KL-cuts produce good partitioning from a logic point of view, as the number of inputs K for each KL-cut tends to be reduced. The fact that KL-cuts are computed on AIGs without explicit inverters helps to preserve the Boolean relation among signals.

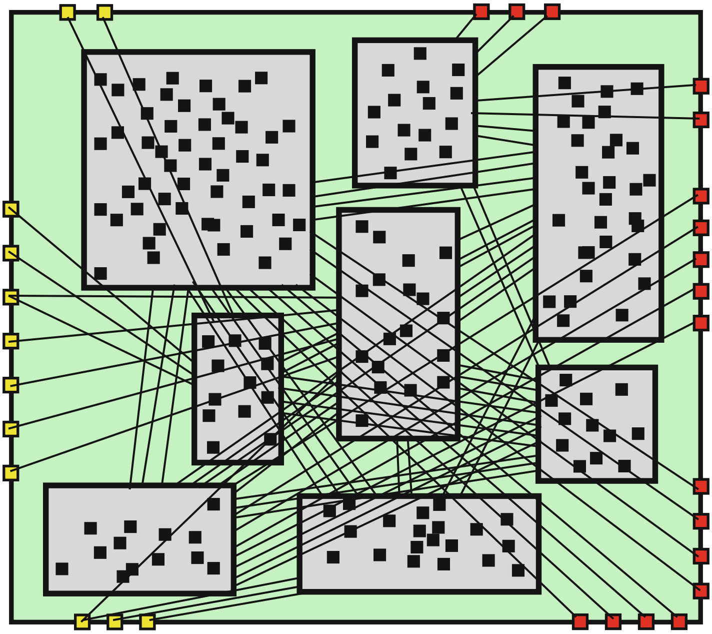


Fig. 4.20 A circuit partitioned into KL-cuts (clusters of AIG nodes): the connections among the clusters and among the I/O pins are also known, but the KL-cuts have not yet been placed

6.4 Placement of Logic Computation KL-Cuts

The placement of KL-cuts is done to produce a KL-cut PAIG. Notice that this placement will assign positions to KL-cuts and to the FFs. This placement is done in such a way to minimize the wirelengths of the connections among KL-cuts, as shown in Fig. 4.21. Notice that the placement of KL-cuts is done to favor interconnects among KL-cuts and I/O pins, the interconnects internal to KL-cuts are ignored during to this step, as KL-cuts are viewed as AIG node containers.

6.5 Physical Design of Global Signal Distribution 1L-Cuts

A signal distribution 1L-cut has one input (from the KL-cut generating the signal) and one or more outputs (to the consumer KL-cuts). For each signal distribution cut, an inverter/buffers topology is generated, with inverter/buffers placed and routed to adequately distribute signals. Notice that the placement and routing is very sparse, using either back-end-of-line (BEOL) or middle-end-of-line (MEOL) long wires to interconnect them. This idea was illustrated in Fig. 4.18, where it is shown that the physical design of a global signal distribution cut is a sparse tree of inverters/buffers.

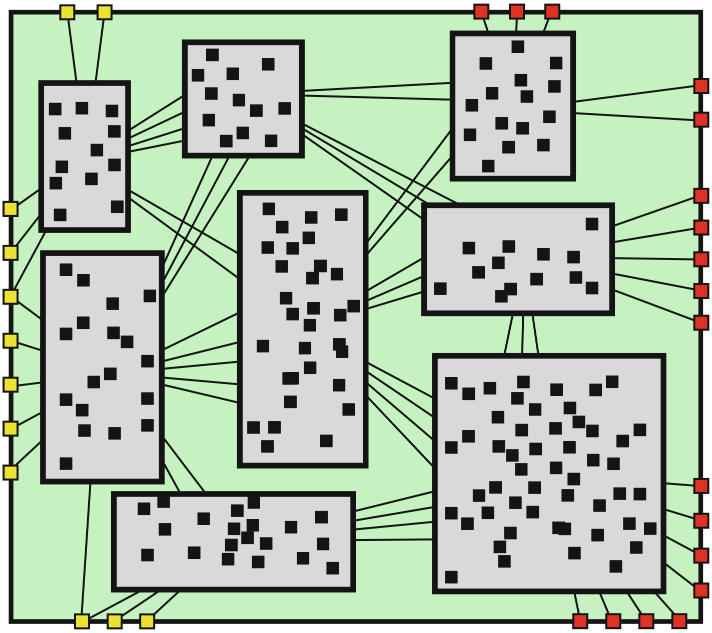


Fig. 4.21 Placement of KL-cuts considering interconnections among KL-cuts and I/O pins

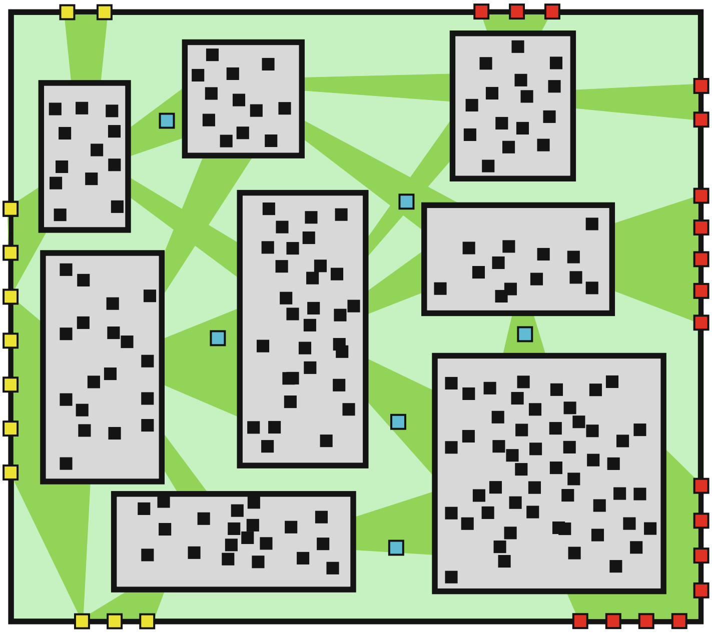


Fig. 4.22 The physical design of global signal distribution 1L-cuts involves the synthesis of several signal distribution cuts (*darker green*)

Figure 4.22 illustrates that the physical design of global signal distribution cuts involves the physical design of several simultaneous cuts (darker green in Fig. 4.22), each of them being a sparse signal distribution tree, as illustrated in Fig. 4.18b. Obviously, the routing of these simultaneous trees compete for routing resources. Once the physical design of global signal distribution cuts is done, the detailed routing among the clusters and among the I/O pins is fully determined, as shown in Fig. 4.23.

6.6 Generate Partial SDCs for Logic Computation KL-Cuts

The physical design of global signal distribution 1L-cuts defines the BEOL/MEOL wires, resulting in a unique feature of the proposed flow, as BEOL/MEOL long wires are defined before FEOL short wires. After the synthesis of signal distribution trees, the long BEOL/MEOL wires with large capacitance are known and accurately characterized. These wires will not change further in the flow, leading to a highly convergent flow.

Once the global signal distribution cuts are physically designed, it is possible to compute local SDC files that express very accurately the delays associated with the physical topology of each 1L-cut. From this, it is possible to compute SDCs for each

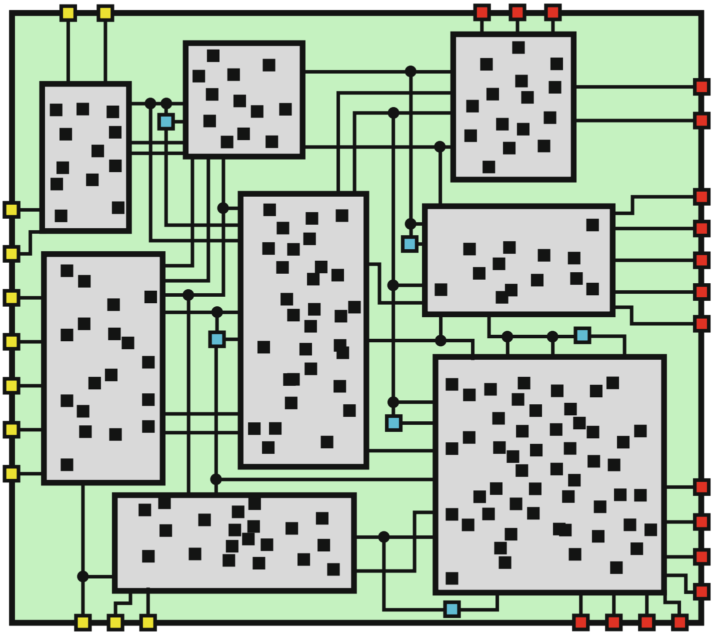


Fig. 4.23 After the physical design of global signal distribution cuts, the detailed routing of global signals is known, allowing to compute delay for long wires. Notice that for didactic purposes, the wires were drawn in between the logic KL-cuts. However, in real designs the logic KL-cuts will have no space in between and the signal distribution will ran on top of the logic KL-cuts

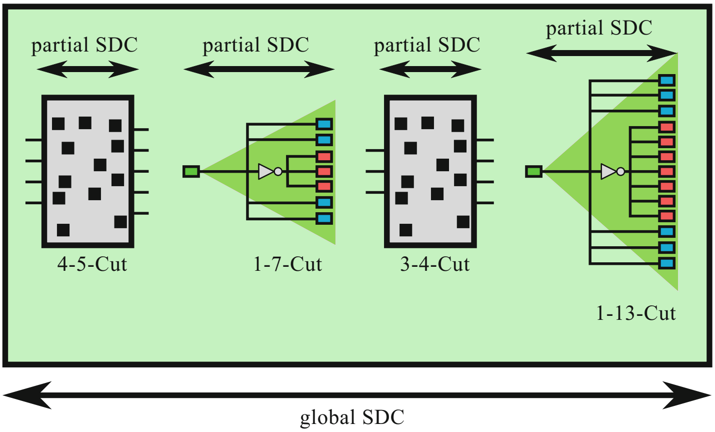


Fig. 4.24 Timing budget: a global SDC is partitioned into a set of partial SDCs, such that global design constraints are still met if partial constraints are met

KL-cut so that the combined local SDCs respect the global SDC constraints. This idea is shown in Fig. 4.24.

6.7 Logic Synthesis of Logic Computation KL-Cuts

As local SDCs are produced for each KL-cut, the logic synthesis of the individual KL-cuts can be done locally according to the associated SDC files [12, 26]. Notice that Boolean techniques [13, 29] can be applied in more demanding regions of the circuit, in such a way to completely restructure the logic to adapt to global requirements of the SDC. Figure 4.25 shows the circuit of Fig. 4.23 after logic synthesis is performed. Notice that the routing of long wires is identical, in this sense Figs. 4.23 and 4.25 are almost identical. The difference is that the structure of the logic inside the KL-cuts has been changed. Notice that this is a local transformation by nature, but it is made to comply with partial SDCs that contain very accurate information about long wires, and that reflect global requirements.

6.8 Physical Design of Logic Computation KL-Cuts

After the logic in the KL-cuts has been restructured to better comply with local SDCs, each KL-cut can be physically designed. As local SDCs are produced for each KL-cut, the physical design of the individual KL-cuts can be done locally

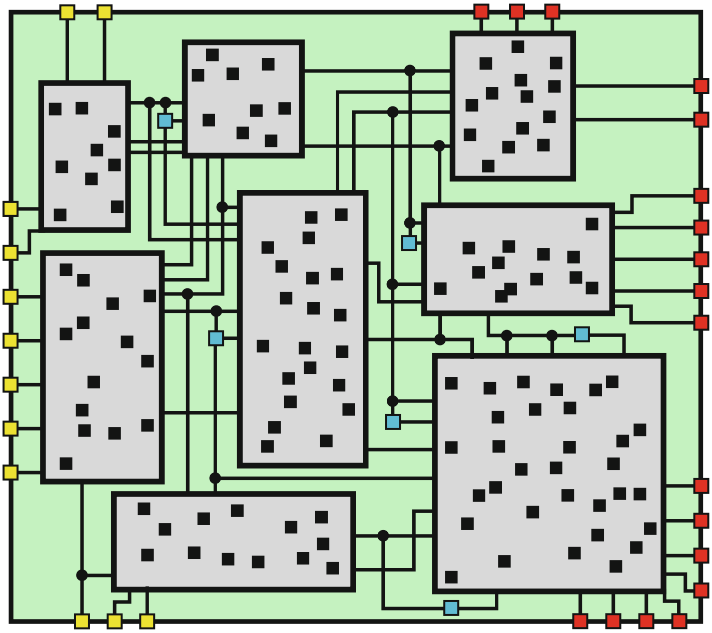


Fig. 4.25 Logic synthesis of logic computation KL-cuts. Notice that the content of the logic KL-cuts has changed with respect to Fig. 4.23

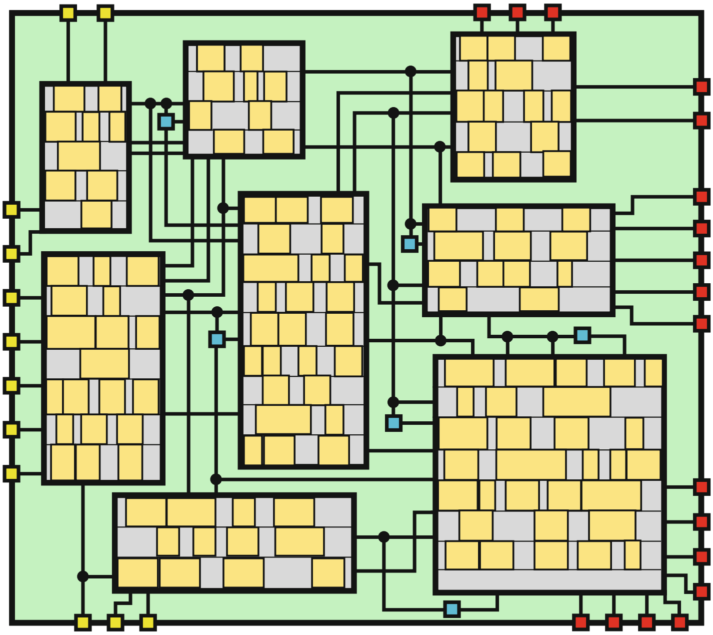


Fig. 4.26 Physical design of logic computation KL-cuts. Notice that the KL-cuts were locally placed and routed

according to the associated SDC files. This consists in placing and routing the cells that compose the KL-cut, as shown in Fig. 4.26. This operation is local by nature, as the KL-cuts are clusters of logic that are placed inside the physical boundaries of the KL-cut, in a very compact way. The internal wires are, by consequence, routed with short wires. These are either front-end-of-line (FEOL) or MEOL wires that do not compete with global signal distribution routing resources. This way, long wires (and large capacitances) are not introduced to modify assumption previously made, resulting in a convergent flow.

6.9 Considerations on Timing Closure, Sinal Distribution, and Logic Computation

During the logic and physical design process of the signal distribution cuts and the logic KL-cuts, the local SDC specifications are produced and followed to guide the synthesis, such that the global SDC constraints are respected, as previously shown in Fig. 4.24. All signal distribution cuts are physically designed before logic cuts, allowing to update the design constraints for signal distribution cuts to reflect the final design instead of initial specification. This idea is illustrated in Fig. 4.27. Slack can be recomputed and propagated to modify design constraints of neighbor logic KL-cuts.

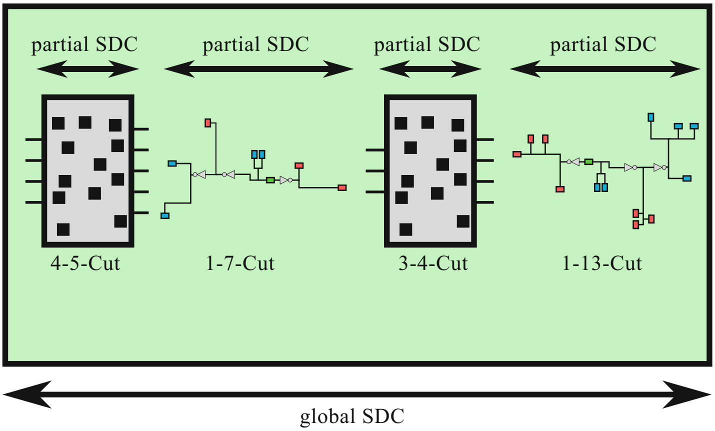


Fig. 4.27 Timing budget: as signal distribution cuts are physically designed first, their associated SDCs can be modified to accurately reflect the effect of long wires. At this moment the topology of the long wires is known and frozen for the remainder of the flow

After the (global and partial) design constraints are updated, the synthesis of logic KL-cuts is performed. Timing closure can be achieved by applying highly Boolean methods [13, 29] to reorganize the local structure of logic KL-cuts. A very positive aspect of the proposed methodology is that the logic inside KL-cuts clusters have their nodes placed densely, using short MEOL/FEOL wires, as shown in Fig. 4.28. This way, results will not be affected due to the insertion of long BEOL wires, as these were defined and frozen during the synthesis of signal distribution cuts (Fig. 4.27). This way, the power of logic restructuring using Boolean methods can be unleashed without the fear that long wires introduced later will invalidate assumptions made during logic synthesis.

The use of partial SDCs for different types of cuts is also very important and helpful for design diagnosis. Problems with design constraints in logic KL-cuts will be corrected with logic restructuring techniques. Problems with design constraints in signal distribution KL-cuts will be corrected with signal distribution techniques. This is a paradigm shift, allowing to achieve a better integration, as the flow is divided into signal distribution and signal computation. As a consequence, both signal distribution and signal computation can benefit from tasks pertaining to logic synthesis and physical design. This way, the emphasis is shifted from what is logic synthesis and what is physical design. The focus is shifted to performing logic synthesis and physical design of signal distribution circuits before performing logic synthesis and physical design of logic computation circuits. The question is not what tasks of Sect. 4 belong to logic synthesis or to physical design. The new question is how these tasks can help with the synthesis of signal distribution 1L-cuts and with the synthesis of logic KL-cuts.

Signal distribution is done mainly through sparse trees of inverters and buffers routed with BEOL/MEOL wires. Signal distribution tasks include inverter/buffer

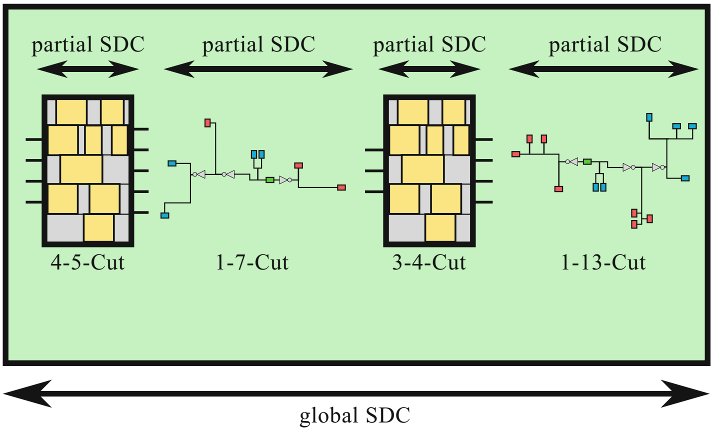


Fig. 4.28 Timing budget: the physical synthesis of KL-cuts is made only after the topology of long wires was already computed

sizing (Sect. 4.1), inverter/buffer Vt swapping (Sect. 4.2), inverter/buffer movement (Sect. 4.3), layer assignment (Sect. 4.4), buffering long nets (Sect. 4.5), buffer deletion (Sect. 4.6), buffering nets to reduce fanout (Sect. 4.7), inverter/buffer cloning (Sect. 4.9), and balancing inverter/buffer trees (similar to Sect. 4.10). Notice that these tasks do not need to be performed individually, but they provide the type of change that can impact positively the signal distribution circuitry.

Logic KL-cuts will be implemented as densely placed groups of interconnected cells using mainly short MEOL/FEOL wires. The synthesis of logic KL-cuts include gate sizing (Sect. 4.1), Vt swapping (Sect. 4.2), buffering nets to reduce fanout (Sect. 4.7), pin swapping (Sect. 4.8), cloning (Sect. 4.9), balancing logic trees (Sect. 4.10), compostion/decomposition (Sect. 4.11), MUX decomposition (Sect. 4.12), and inverter absorption (Sect. 4.13). Notice that these tasks do not need to be performed individually, but they provide the type of change that can impact positively the logic KL-cut rewriting. Highly Boolean methods [13, 29] can also be applied to reorganize the local structure of logic KL-cuts.

7 Conclusion

We proposed an innovative flow for designing digital integrated circuits. This flow can bring technology information earlier in the design cycle. A unique feature of the proposed flow is that long global wires are defined first and have the topology frozen to respect global timing. With long wires defined, the local clusters of logic are synthesized. During the process, partial design constraints are produced (updated) and followed to guide the synthesis, such that the global design constraints are respected. This is a paradigm shift to a VLSI design flow based on the concepts of signal distribution and logic computation. The proposed flow allows the power of logic restructuring using Boolean methods to be unleashed without the fear that long wires introduced later during physical design would invalidate assumptions made during logic synthesis.

References

1. C. Alpert, When logic synthesis met physical synthesis, *Invited Talk at the 22nd Workshop on Logic and Synthesis* (2013)
2. C.J. Alpert, J. Hu, S.S. Sapatnekar, P.G. Villarrubia, A practical methodology for early buffer and wire resource allocation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **22**(5), 573–583 (2003)
3. C.J. Alpert, D.P. Mehta, S.S. Sapatnekar, *Handbook of Algorithms For Physical Design Automation* (CRC Press, Boca Raton, 2008)
4. M.R. Berkelaar, J.A. Jess, Gate sizing in MOS digital circuits with linear programming, in *Proceedings of the Conference on European Design Automation* (IEEE Computer Society Press, Los Alamitos, 1990), pp. 217–221
5. C.L. Berman, J.L. Carter, K.F. Day, The fanout problem: from theory to practice, in *Proceedings of the decennial Caltech Conference on VLSI on Advanced Research in VLSI* (MIT Press, Cambridge, 1989), pp. 69–99
6. K. Chang, D.C. Du, Efficient algorithms for layer assignment problem. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **6**(1), 67–78 (1987)
7. C.W. Chang, M.F. Hsiao, B. Hu, K. Wang, M. Marek-Sadowska, C.K. Cheng, S.J. Chen, Fast postplacement optimization using functional symmetries. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **23**(1), 102–118 (2004)
8. K.-H. Chang, I.L. Markov, V. Bertacco, Postplacement rewiring by exhaustive search for functional symmetries. ACM Trans. Des. Autom. Electron. Syst. **12**(3), 32:1–32:21, Article 32 (2008)
9. O. Coudert, Gate sizing for constrained delay/power/area optimization. IEEE Trans. Very Large Scale Integr. Syst. **5**(4), 465–472 (1997)
10. K.R. Dai, W.H. Liu, Y.L. Li, NCTU-GR: Efficient simulated evolution-based rerouting and congestion-relaxed layer assignment on 3-d global routing. IEEE Trans. Very Large Scale Integr. Syst. **20**(3), 459–472 (2012)
11. S. Devadas, A. Ghosh, K.W. Keutzer, *Logic Synthesis* (McGraw-Hill, New York, 1994)
12. M. Elbayoumi, M. Choudhury, V. Kravets, A. Sullivan, M. Hsiao, M. Elnainay, Tacue: a timing-aware cuts enumeration algorithm for parallel synthesis, in *Proceedings of the 51st Annual Design Automation Conference, DAC '14* (2014)
13. T. Figueiró, R.P. Ribas, A.I. Reis, Constructive AIG optimization considering input weights, in *2011 12th International Symposium on Quality Electronic Design* (2011)
14. S.H. Gerez, *Algorithms for VLSI Design Automation*, vol. 8 (Wiley, New York, 1999)
15. G.D. Hachtel, F. Somenzi, *Logic Synthesis and Verification Algorithms* (Springer, New York, 2006)
16. S. Hassoun, T. Sasao, *Logic Synthesis and Verification*, vol. 654 (Springer, New York, 2012)
17. J. Hu, A.B. Kahng, S. Kang, M.C. Kim, I.L. Markov, Sensitivity-guided metaheuristics for accurate discrete gate sizing, in *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (IEEE, 2012), pp. 233–239
18. A.B. Kahng, J. Lienig, I.L. Markov, J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure* (Springer, New York, 2011)
19. A.B. Kahng, S. Kang, H. Lee, I.L. Markov, P. Thapar, High-performance gate sizing with a signoff timer, in *Proceedings of the International Conference on Computer-Aided Design* (IEEE Press, Piscataway, 2013), pp. 450–457

20. S.P. Khatri, K. Gulati, *Advanced Techniques in Logic Synthesis, Optimizations and Applications* (Springer, New York, 2011)
21. J.M. Kleinhans, G. Sigl, F.M. Johannes, K.J. Antreich, Gordian: VLSI placement by quadratic programming and slicing optimization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **10**(3), 356–365 (1991)
22. J. Lillis, C.K. Cheng, T.T. Lin, Algorithms for optimal introduction of redundant logic for timing and area optimization, in *1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World, ISCAS'96*, vol. 4 (IEEE, 1996), pp. 452–455
23. S.K. Lim, *Practical Problems in VLSI Physical Design Automation* (Springer, Dordrecht, 2008)
24. Y. Liu, J. Hu, A new algorithm for simultaneous gate sizing and threshold voltage assignment. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **29**(2), 223–234 (2010)
25. L. Machado, M. Martins, V. Callegaro, R.P. Ribas, A.I. Reis, Kl-cut based digital circuit remapping, in *Proceedings of the NORCHIP 2012* (2012)
26. L. Machado, M.G.A. Martins, V. Callegaro, R.P. Ribas, A.I. Reis, Iterative remapping respecting timing constraints, in *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 236–241 (2013)
27. I.L. Markov, J. Hu, M.C. Kim, Progress and challenges in VLSI placement research. Proc. IEEE **103**(11), 1985–2003 (2015)
28. O. Martinello, F.S. Marques, R.P. Ribas, A.I. Reis, Kl-cuts: a new approach for logic synthesis targeting multiple output blocks, in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 777–782 (2010)
29. M.G.A. Martins, L. Rosa, A.B. Rasmussen, R.P. Ribas, A.I. Reis, Boolean factoring with multi-objective goals, in *2010 IEEE International Conference on Computer Design*, pp. 229–234 (2010)
30. M. Martins, J.M. Matos, R.P. Ribas, A. Reis, G. Schlinker, L. Rech, J. Michelsen, Open cell library in 15nm freepdk technology, in *Proceedings of the 2015 Symposium on International Symposium on Physical Design, ISPD '15* (2015)
31. G.D. Micheli, *Synthesis and Optimization of Digital Circuits* (McGraw-Hill, New York, 1994)
32. R. Murgai, On the global fanout optimization problem, in *1999 IEEE/ACM International Conference on Computer-Aided Design*. Digest of Technical Papers (Cat. No.99CH37051) (1999)
33. R. Nair, C.L. Berman, P.S. Hauge, E.J. Yoffa, Generation of performance constraints for layout. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **8**(8), 860–874 (1989). https://doi.org/10.1109/43.31546
34. G.J. Nam, J.J. Cong, *Modern Circuit Placement: Best Practices and Results* (Springer, Boston, 2007)
35. I. Neumann, D. Stoffel, H. Hartje, W. Kunz, Cell replication and redundancy elimination during placement for cycle time optimization, in *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design* (IEEE Press, Piscataway, 1999), pp. 25–30
36. J. Qian, S. Pullela, L. Pillage, Modeling the "effective capacitance" for the RC interconnect of CMOS gates. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **13**(12), 1526–1535 (1994)
37. R. Rudell, Logic synthesis for vlsi design, Ph.D. thesis, University of California, Berkeley (1989)
38. L. Scheffer, L. Lavagno, G. Martin, *EDA for IC Implementation, Circuit Design, and Process Technology* (CRC Press, Hoboken, 2006)
39. E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, A.L. Sangiovanni-Vincentelli, SIS: a system for sequential circuit synthesis. Technical Report No. UCB/ERL M92/41. EECS Department, University of California, Berkeley (1992)
40. S. Shah, A. Srivastava, D. Sharma, D. Sylvester, D. Blaauw, V. Zolotov, Discrete VT assignment and gate sizing using a self-snapping continuous formulation, in *IEEE/ACM International Conference on Computer-Aided Design, ICCAD-2005* (IEEE, 2005), pp. 705–712

41. A. Srivastava, R. Kastner, C. Chen, M. Sarrafzadeh, Timing driven gate duplication. IEEE Trans. Very Large Scale Integr. Syst. **12**(1), 42–51 (2004)
42. L.P. Van Ginneken, Buffer placement in distributed RC-tree networks for minimal Elmore delay, in *IEEE International Symposium on Circuits and Systems, 1990* (IEEE, 1990), pp. 865–868
43. L.T. Wang, Y.W. Chang, K.T.T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test* (Morgan Kaufmann, Burlington, 2009)
44. J. Werber, D. Rautenbach, C. Szegedy, Timing optimization by restructuring long combinatorial paths, in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-aided Design* (2007)

Chapter 5
Identifying Transparent Logic in Gate-Level Circuits

Yu-Yun Dai and Robert K. Brayton

1 Introduction

In hardware, control logic regulates the data flow and dictates circuit functionalities. A logic that simply moves data from one part of a circuit to another without modifying it can be referred to as *transparent*. Another category transforms data by some word-level operator, e.g. a bit-vector operator defined in Verilog. A third category, control, determines which data is moved and when, or which operation is applied and when. Efficient recognition of such logic can benefit circuit verification, e.g. [4] as a guide to abstraction. To identify word-level operators in a gate-level circuit, it is crucial to find words and locate the boundaries (inputs/outputs) of arithmetic operators [8].

The basic example of transparent logic is a multiplexer (MUX) structure, which selects from several data signals and forwards it unaltered towards the outputs. Identification of MUXes can be performed over gate-level circuits very quickly using structural matching, but it can be unreliable, especially if logic synthesis has been applied.

In this paper, we focus on functional approaches which do not depend on the actual gate-level structure of the circuit. These can augment structure methods and provide a much more reliable technique as we show in the experiments.

Functional methods, which rely only on functional dependencies, have been generally used to augment structural approaches. Examples are:

– Li and Subramanyan et al. [6, 10] identified internal words based on *bitslice aggregation* (functional approach) and *shapehashing* (structural approach). The candidate words found were used as boundaries of operators for further recogni-

Y.-Y. Dai (✉) • R.K. Brayton
Department of EECS, University of California, Berkeley, Berkeley, CA, USA
e-mail: yunmeow@berkeley.edu

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_5

tion. However, they did not recognize found boundaries as inputs or outputs of operators before identifying functionality.

- Li et al. [6, 7] identified functional operators in gate-level circuits, based on an existing library of blocks. Word-level information at the primary inputs was assumed available, but in many applications this information is not known.
- Sterin et al. [8] extracted word-level operators functionally, given a library of operators and a slice of logic containing inputs and outputs of such operators. The possible location and ordering of the inputs and outputs of an operator and word-level information were not required. However, the slice of logic cannot include transparent logic, based on the algorithm.

We present methods to identify *functional transparent logic*. This is inherited from *functional isomorphism*. Using this approach, an algorithm to identify words, word-level operator boundaries, and control logic in gate-level circuits is proposed and applied to a variety of test cases. Indeed, we can rewrite a gate-level circuits hierarchically (i.e., hierarchical Verilog format) with the recognized logic as sub-circuits. Once operator boundaries are (roughly) located, techniques like [8] can be used to identify the precise location of the operators and their functionalities.

The chapter is organized as follows. Section 2 introduces *functional* isomorphism. In Sect. 3, we describe the definition and propagation of *transparent* logic. Proposed algorithms for identifying transparent logic are given in Sect. 4. Some practical challenges of identifying transparent logic are discussed in Sect. 5. Experimental results are shown in Sect. 6, while Sect. 7 concludes this chapter.

2 Overview

Roughly, a transparent path in a circuit has width n and a set of controls $\{s^i\}$, which when evaluated appropriately at a minterm $s^i = m_{s^i}$, moves a data-word (width n) from the beginning of the path to the end. Such paths can fork and join in the circuit, and can begin and end at a set of inputs, outputs, or internal signals. A path is maximal if there is no transparent path that can extend it. The terminals of maximal transparent paths are of interest because they likely delineate the input or output of an operator, e.g. an arithmetic function.

A sink terminal can have many source terminals. Each signal in a sink terminal is a Boolean function of (a) data signals $D_k = \{d_k^j\}$ in the source terminals and (b) the set of associated controls $\{s^i\}$ of transparent segments of any path from source to sink. Such a set of functions at a sink forms an NPN equivalence class (or equivalently an NPN isomorphism class). Each sink bit function typically (with some exceptions) looks like $f_k = \sum_{j \in D_k} (\prod_{i \in \text{path}} m_{s^i}) d_k^j$. The isomorphism between the inputs of any two signals f_p and f_q (where $p, q \in [1, n]$) in the sink terminal is $d_p^j \leftrightarrow d_q^j$, i.e. different bit positions in the same data word are isomorphically mapped to each other, while control signals s^i are isomorphically mapped into themselves. Each coefficient $(\prod_{i \in \text{path}} m_{s^i})$ is the predicate of the control signals under which a path from source d_k^j to the sink f_k becomes transparent. The predicates are disjoint. It is possible that some bits of a terminal have been inverted, hence NPN equivalence is considered in the subsequent discussions.

Thus, the outputs of a transparent path are a subset of an NPN isomorphism class. The isomorphism helps distinguish between different data-words by factoring out common predicates ($\prod_{i \in \text{path}} m_{s^i}$) in the representative function of the equivalence class.

2.1 NPN Isomorphism

Two graphs, $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, are *isomorphic*, if there exists a bijective mapping, $\mathbf{M}_{12}$: $V_1 \rightarrow V_2$, such that any two vertices u and v are adjacent in G_1, if and only if $\mathbf{M_{12}}(u)$ and $\mathbf{M}_{12}$ (v) are adjacent in G_2 [1]. Two circuits, C_1 and C_2, are isomorphic to each other if their logic gates and connections form two isomorphic graphs, while any gate g of C_1 and the mapped gate $\mathbf{M}_{12}(g)$ in C_2 are the same type. The relation between C_1 and C_2 is called *structural isomorphism*, which has been applied to reverse engineering [5].

In contrast, *functional isomorphism* is a relation between two signals in a circuit. A signal f in a circuit, supported by a set of other signals, S_f, is a Boolean function of these inputs: $f : \mathbf{B}^{|S_f|} \rightarrow \mathbf{B}$, for $\mathbf{B} = \{0, 1\}$.

In the following sections, for a Boolean variable x_i with its polarity p_i, $(x_i)^{p_i}$ represents the function: $p_i = 0 \rightarrow (x_i)^{p_i} \equiv x_i$ and $p_i = 1 \rightarrow (x_i)^{p_i} \equiv \text{inv}(x_i)$.

Definition 1 A pair of Boolean functions $f(x_1, \ldots x_n)$ and $g(y_1, \ldots, y_n)$ are *NPN isomorphic*,[1] if there exists a permutation π of size n and polarities p_{out} and $\{p_1, \ldots, p_n\} \in \mathbf{B}^n$ such that

$$f(x_1, \ldots, x_n) = g^{p_{\text{out}}}(x_{\pi(1)}^{p_1}, \ldots, x_{\pi(n)}^{p_n}) \tag{5.1}$$

i.e., g can be made equivalent to f by selectively negating inputs, permuting inputs, and negating the output. The implied isomorphic mapping between the supports of g and f is $\{y_i, x_{\pi(i)}^{p_i}\}$ and p_i is said to be the relative polarity between inputs y_i and $x_{\pi(i)}$.

A set of signals in a circuit, in which every pair is functionally NPN isomorphic is called an *NPN isomorphism class*.

[1] Or Negation-Permutation-Negation (NPN) equivalent.

2.2 Composition of NPN Isomorphism

Although improved methods for computing NPN equivalence can be found in Soekin et al. [9], this calculation can still be time-consuming. This effort can be reduced immensely by proving NPN isomorphism on smaller logic blocks and composing proved classes to obtain larger ones. Larger classes help extend paths of transparency (discussed in Sect. 3) in a circuit and to more reliably find transparency boundaries, and hence the input/output boundaries of word-level operators.

The following discussion details when compositions lead to larger NPN isomorphisms.

Definition 2 (Polar Consistency) Let $(f(s), g(t))$ be a pair of NPN isomorphic functions with sets of supports $s = \{s_i\}$ and $t = \{t_j\}$, respectively. Suppose each pair of mapped input supports $s_i \leftrightarrow t_j$ are NPN isomorphic functions, i.e. $s_i(x)$ is NPN isomorphic to $t_j(y)$. Let p^{ij}_{out} be the relative output polarity between $s_i(x)$ and $t_j(y)$, and p_{ij} be the relative input polarity between inputs s_i and t_j in the NPN isomorphism between $f(s)$ and $g(t)$. The compositions $f(s(x))$ and $g(t(y))$ are *polar consistent*, if $p^{i\pi(i)}_{\text{out}} = p_{i\pi(i)}$, where π is the permutation in the isomorphism mapping of $(f(s), g(t))$.

Theorem 1 *The compositions of $(f(s(x)), g(t(y)))$ are polar consistent if and only if $f(s(x))$ and $g(t(y))$ are NPN isomorphic.*

3 Transparent Logic

As already stated, the identification of maximal *transparent logic* can be used to identify input/output boundaries of arithmetic operators.

3.1 Transparent Words

Intuitively, a *transparent word* is a set of signals, $\{w^k\}$, with supports, $\{S^k\}$, where under some evaluation of $\cap^k S^k$ (*common control*), $\{w^k\}$ is equivalent to a subset (*data-word*) of $\cup^k S^k$. In other words, the control evaluation makes the word transparent from some input data-word.

Example An m-bit word from a set of 2-to-1 multiplexers (MUX) controlled by the same selector signal s,

$$C[m-1:0] = s?A[m-1:0] : B[m-1:0], \tag{5.2}$$

comprises a transparent word C, where $\forall j \in [0, m-1]$, $(C[j] = sA[j] + s'B[j])$. For this case, word C is transparent from word A or word B, depending on the value assigned to s.

Definition 3 Functions $W = \{w^k | k \in [1, m]\}$ of an NPN isomorphism class comprise an m-bit *transparent word*, if:

1. Each function $w^k : \mathbf{B}^{S^k} \rightarrow \mathbf{B}$, has support $S^k = (Control, Data^k)$, i.e. *Control* is the set of common signals, and each bit of *Control* is isomorphically mapped into itself.
2. The formula (3) is *True*, where m_c is a minterm of *Control*, $\equiv$ denotes functional equivalence, and $w^k_{m_c}$ denotes the co-factor of function $w^k(Control, Data^k)$ with respect to m_c.

$$(\exists_{m_c} \forall_k \exists_{d_i^k \in Data^k} \exists_{p_i^k} (w^k_{m_c}(Data^k) \equiv (d_i^k)^{p_i^k})). \tag{5.3}$$

3. For any $(w^x, w^y) \in W$, the associated isomorphic support mapping $\mathbf{M_{xy}}$, satisfies $\mathbf{M_{xy}}(Data^x) = Data^y$.

Thus a transparent word W is conditionally (by m_c) equivalent to an input data word $[(d_i^1)^{p_i^1}, \ldots, (d_i^m)^{p_i^m}]$. Based on the above definition, the vector of conditionally equivalent data support bits that have a common condition m_c, $D_i = \{d_i^1, \ldots, d_i^m\}$ is called an *input word*.

Given a transparent word, $W = \{w_k\}$, with the corresponding support partitions $\{(Control, Data^k)\}$, the entire support set of W can be partitioned into **Control** and $\mathbf{Data}^W = \bigcup^i D_i$. The definition of transparent words can be restated as follows:

Definition 4 A transparent word W is a set of NPN isomorphism functions supported by control **Control** and data $\mathbf{Data}^W = \bigcup^i D_i$, such that the following formula is *True*:

$$\forall_{D_i \in \mathbf{Data}^W} \exists_{m_c} \exists_{P_i} (W_{m_c}(\mathbf{Data}^W) \equiv (D_i)^{P_i}), \tag{5.4}$$

where P_i set of polarity bits for D_i.

Although, for an input word D_i, there could be multiple minterms of *Control* satisfying Formula (4), the assignments of $m_c \in Control$ for different D_is are disjoint.

Example Consider Eq. (5.2): for each $C[j]$, the support set $\{s, A[j], B[j]\}$ can be partitioned into $Data^j = \{A[j], B[j]\}$ and $Control = \{s\}$, such that $(s = 1) \Rightarrow (C[j] = A[j])$ and $(s = 0) \Rightarrow (C[j] = B[j])$. Hence a common (control) assignment applied to all bits of the transparent word makes them transparent from the corresponding supports simultaneously. The supports of C can be partitioned into $\mathbf{Data}^C \equiv \{A[m-1:0], B[m-1,0]\}$, and $\mathbf{Control} \equiv \{s\}$.

Since negations of some bits of transparent words might occur during synthesis, it seems reasonable to consider the logic still as "transparent." Note that in the example: $C[j] = sA[j] + s'B[j]$ the negation of bit $C[j]$ can be done by negating the data inputs, $A[j]$ and $B[j]$:

$$\begin{aligned} inv(C[j]) &= inv(sA[j] + s'B[j]) \\ &= s\, inv(A[j]) + s' inv(B[j]). \end{aligned} \tag{5.5}$$

but C (with some phase changes) can still be considered transparent from A and B because the assignments to the control bits are unchanged.

In Sects. 3.2 and 3.3, as we compose transparent sections to form a larger transparent path, we will need to resolve cases where only some bits of a transparent word are negated. However, for composing transparencies to find larger ones, it is required that the polarities of the inputs and outputs are consistent. This can be done by negating some of the inputs of the path (using NPN isomorphism) to get a compatible polarity at the output that feeds into another transparent word.

Theorem 2 *Given a transparent word W, the negation of any output bit w^k can be done by negating the corresponding input data support bits, without changing any control assignment.*

The upshot is that when finding another transparent section of logic and composing it to extend a transparent path, this can *always* be done simply by negating the inputs to get compatible polarities at the point of composition.

3.2 Composition of Transparency

Similar to the composition of NPN isomorphism, larger transparent functions are frequently created by composing smaller transparent blocks.

Example In Fig. 5.1, word C is transparent from A and B under the control of s_1, while a second transparent block consists of word E, transparent from C and D under the control of s_2. Thus $(s_1 = 1, s_2 = 1) \rightarrow E \equiv A$, while $(s_1 = 0, s_2 = 1) \rightarrow E \equiv B$ i.e. transparency of E from A and B is obtained by composing of smaller transparent blocks. If some bits of C are negated before feeding into the MUXes controlled by

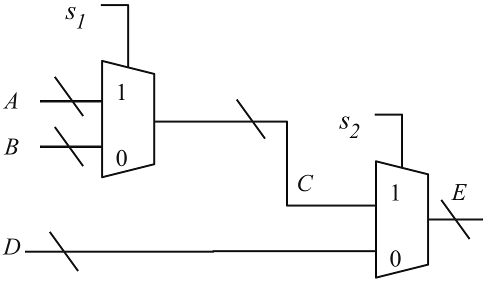


Fig. 5.1 A transparent word can be implemented by composing smaller transparent words

s_2, the composition can be done by pushing the negation to the corresponding bits of A and B to maintain the polar consistency.

Definition 5 Let $\mathbf{W} = \{W^k(X) | k = [1, n]\}$ be a set of n m-bit transparent words, and let $Y = \{y^j | j = [1, m]\}$ be another transparent word with support $Data^Y = \mathbf{W} \bigcup V$ and $Control^Y$. Suppose each input word of Y is exactly one transparent word in $\mathbf{W}$ or one word in V. The set of compositions,

$$Z = \{z^j\} = \{y^j(\mathbf{W}(X), V, Control^Y)\} \tag{5.6}$$

form a *compound word*, and are denoted as $Z = Y \circ \mathbf{W}$.

Theorem 3 *Assume Y is a transparent word and $\mathbf{W}$ is a set of transparent words. Let $\{\alpha_i^k\}$ be the set of minterms of $Control^k$, which enable W^k to be transparent from an input word $x_i^k \in Data^k$, and $\{\beta^k\}$ be the set of minterms of $Control^Y$ for ($Y \equiv W^k$). Using the notation:*

$$Control^Z = Control^Y \bigcup [\cup^k Control^k],$$

$$Data^Z = V \bigcup [\cup^k Data^k],$$

a compound word, $Z \equiv Y \circ \mathbf{W}$ is a transparent word controlled by $Control^Z$ if

$$\forall_k \forall_i (\{\hat{\alpha_i^k}\} \cap \{\hat{\beta^k}\} \neq \emptyset) \tag{5.7}$$

is True*, where $\{\hat{\alpha_i^k}\}$ and $\{\hat{\beta^k}\}$ are $\{\alpha_i^k\}$ and $\{\beta^k\}$ extended to cubes of the larger space of $Control^Z$, respectively.*

Proof

1. Based on Theorems 1 and 2, Z can be an NPN isomorphism class by flipping the polarities of W^kwhenever its output polarity is not consistent with the input polarities of y^k.
2. Because Y is a transparent word, for each input word in V, there must exist an assignment of $Control^Y$ to enable the transparency from V.
3. Conditions satisfying Formula (5.7) imply that for each input word x_i^k of W^k, there exists an assignment of $Control^Z$ such that a) W^k is transparent from x_i^k, b) Y is transparent from W^k, and c) Y is transparent from x_i^k. Therefore, $Z \equiv Y \circ \mathbf{W}$ is a transparent word with ($Control^Z$, $Data^Z$) as control and data supports.

3.3 Propagation of Transparency

Example Figure 5.2 illustrates how a longer transparency can be composed from nontransparent sections of logic. C is transparent from A when $s_1 = 1$, and D is transparent from B when $s_2 = 1$, but the logic block from C and D to E is not

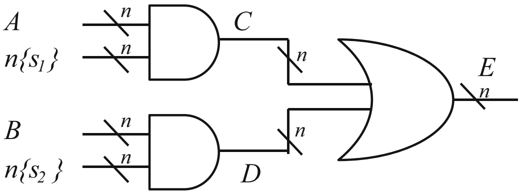


Fig. 5.2 A longer transparent word may be composed of smaller transparent words and an NPN isomorphism class

transparent (there is no common control support for each bit of E). However, E is transparent from A when $(s_1 = 1, s_2 = 0)$, while $(s_1 = 0, s_2 = 1)$ makes E transparent from B.

When a transparent function block is composed of non-transparent sections, it is called *propagation of transparency*. The conditions when this can happen are stated in the following.

Definition 6 (Proceeding Word) Let $\mathbf{W}$ be a set of n m-bit transparent words, and let $Y(\mathbf{W}) = \{y^j(\mathbf{W})\}$ be an NPN isomorphism class. Suppose each y^j is supported by exactly one bit of each W^k, and the isomorphically mapped supports of y^j are always from the same word of $\mathbf{W}$.

Theorem 4 *Assume Y and $\mathbf{W}$ are as in Definition 6 and the supports of W^k are $supp(W^k) = (Control^k, Data^k)$. Let $\{\alpha_i^k\}$ be the set of minterms of $Control^k$ which cause $(W^k \equiv x_i^k)$, and $\{\beta^k\}$ be minterms of $\cup^k Control^k$ which cause $(Y \equiv W^k)$. Using $Control^Z = \cup^k Control^k$, and $Data^Z = \cup^k Data^k$, a proceeding word, $Z \equiv Y \circ \mathbf{W}$, is a transparent word controlled by $Control^Z$ if*

$$\forall_k \forall_i (\{\hat{\alpha}_i^k\} \cap \{\beta^k\} \neq \emptyset), \tag{5.8}$$

where $\{\hat{\alpha}_i^k\}$ refers to $\{\alpha_i^k\}$ extended to cubes of $Control^Z$.

Proof

1. Similar to the proof of Theorem 3, Z can be an NPN isomorphism class by flipping the polarities of W^k if needed.
2. For each input word x_i^k in $Data^Z$, Formula (5.8) implies that there exists an assignment of $Control^Z$, such that $W^k \equiv x_i^k$, $Y \equiv W^k$, and thus, $Y \equiv x_i^k$, implying $Z \equiv Y \circ \mathbf{W}$ is a transparent word with $(Control^Z, Data^Z)$ as control and data supports.

Example In Fig. 5.2, $\mathbf{W} = (C, D)$ and $\{\hat{\alpha}_1^k\} = s_1 s_2 + s_1 \bar{s}_2$ makes C transparent from A, while $\{\hat{\alpha}_2^k\} = s_1 s_2 + \bar{s}_1 s_2$ makes D transparent from B. $\{\beta^k\} = s_1 \bar{s}_2$ $(\bar{s}_1 s_2)$ causes $E \equiv C$ $(E \equiv D)$. Note that $\{\hat{\alpha}_1^k\} \cap \{\beta^k\} = s_1 \bar{s}_2 \neq \emptyset$ and $\{\hat{\alpha}_2^k\} \cap \{\beta^k\} = \bar{s}_1 s_2 \neq \emptyset$. Thus the conditions for propagation of transparency are met, and therefore E is transparent from A and B.

4 Transparency Identification

The functional approach proposed for transparency identification relies only on dependencies among signals. It can be used to complement a structural approach, leading to a method that is still efficient but with more reliable results.

In general, we want to identify transparent logic anywhere it occurs in the circuit—from inputs to internal words (forward), from internal words to outputs (backward), and between internal words. One general problem is formulated and solved in this chapter: given a combinational circuit, find (1) all (disjoint) transparent logic blocks, specified by corresponding support and output boundaries, (2) transparent words on each output boundary, (3) input words on each support boundary, and (4) assignments of **Control** for moving each input word to the output transparent word.

The proposed algorithm can be decomposed into four parts: (1) collect candidate controls, (2) find transparent words controlled by one signal, (3) find proceeding words, and (4) rearrange proved words.

4.1 Find Transparency with Given Controls

A sub-process, *findTransparency(. . .)*, is developed to identify all transparent words controlled by a set of signals. Figure 5.3 shows the proposed algorithm for this sub-process.

In Line 1, the function *enumerateControls(. . .)* enumerates all possible minterms for the input control set. For each minterm, the function *applyMinterm(. . .)* finds the co-factor of the input circuit. It returns a set of conditionally transparent paths, **Candidates**, where each sink signal is functionally equivalent to the corresponding source. The output of each transparent path must be supported by all signals in

```
Algorithm: Find Transparency
Input:
  Circuit, Controls
Output:
  TransparentWords = (Inputs, Outputs, Minterms)

01. Minterms = enumerateControls(Controls)
02. For each minterm in Minterms
03.   Candidates = applyMinterm(minterm, Circuit)
04. TransparentWords = analyzeWords(Candidates)
05. splitWords(TransparentWords)
06. Return TransparentWords
```

Fig. 5.3 The sub-process for identifying transparent words controlled by a set of signals

Controls. Note that a sink signal can be driven by multiple transparent paths controlled by different minterms of the same set of controls.

In Line 4, the function *analyzeWords(. . .)* examines all candidate paths and merges paths with the same sink signal. Then this function partitions those sink signals into several transparent words. To match the requirement of NPN isomorphism, in each transparent word, all output signals are controlled by an identical set of minterms, and the depths (number of signals along each path, excluding sources) of all transparent paths are the same. Also, if some input bits are primary inputs, while the other mapped bits (under the same set of minterms) are internal signals, they are classified into separate words.

The function *splitWords(. . .)* partitions a word if some output bits of it are primary outputs, while others are internal signals.

4.2 Overall Algorithm Flow

Based on the function in Figs. 5.3 and 5.4 outlines the steps for finding all transparent words and identifying words on support or output boundaries for an input circuit.

```
Algorithm: Functional Approach
Input:
  Circuit
Output:
  TransparentBlocks = (Outputs, Supports, Words)

01. ProvedWords= $\emptyset$
02. CandidateControls = $findHighFanoutSignals$(Circuit)
03. For each $control$ in CandidateControls
04.   NewWords = $findTransparency$(Circuit, $\{control\}$)
05.   ProvedWords $\bigcup$= NewWords
06. For each $word$ in ProvedWords
07.   If $notFullyTransparent$ ($word$)
08.     ControlSets = $findPossibleCombinations$ $word$)
09.     For each $controlSet$ in ControlSets
10.       If $newCombination$($controlSet$)
11.         NewWords = $findTransparency$(Circuit, $controlSet$)
12.         $extend$(ProvedWords, NewWords)
13. $cleanMultiplyDrivenWords$(ProvedWords)
14. $partitionWords$(ProvedWords)
15. $disposeWords$(ProvedWords)
16. TransparentBlocks = $analyzeBlock$ (ProvedWords)
17. Return TransparentBlocks
```

Fig. 5.4 The proposed algorithm for identifying all transparent words in a gate-level circuit

To find control candidates, the function *findHighFanoutSignals(...)* in Line 2 uses the fact that all bits of a transparent word must be controlled by the same condition. It collects all signals with more than three immediate fanouts.

Lines 3–5 find all transparent words controlled by a single signal, including the standard 2-to-1 MUXes and depth-one words.

To recognize proceeding words, Lines 8 to 12 work on words which so far are not fully feeding into transparent paths. For each candidate word, the function *findPossibleCombination(...)* collects its depth-one fanouts and finds other transparent words which also support those fanouts. If the signal dependencies of supporting words and fanouts satisfy **Definition 6**, they may result proceeding words. Note that if the input circuit is an and-inverter graph (AIG), when only depth-one fanouts are considered, each proceeding word can only come from two proved words.

Then the union of the control sets of the two words is a candidate control set. The algorithm executes *findTransparency(...)* if this control set has never been considered before. The newly found words are appended at the end of **ProvedWords** for being examined by the same flow later.

Lines 13 and 14 rearrange all proved transparent words to achieve a legal word dependency graph, in which multiply-driven signals are absent, and all bits of one word are driven by the same set of words. The function *cleanMultiplyDrivenWords(...)* examines every signal driven by more than one transparent path and assigns it to the most *preferred* word. This process favors wider words first, and then deeper ones. The function *partitionWords(...)* partitions each word into smaller words if some bits are supported by different input words. Also, outputs driving different sets of words are grouped into different words.

The function *disposeWords(...)* discards transparent words with bad properties; it excludes all words with fewer than 4 bits after the above processes; it can also discard some depth-one words if they are suspected as *bad* transparent words—more details are discussed in Sect. 5.2.

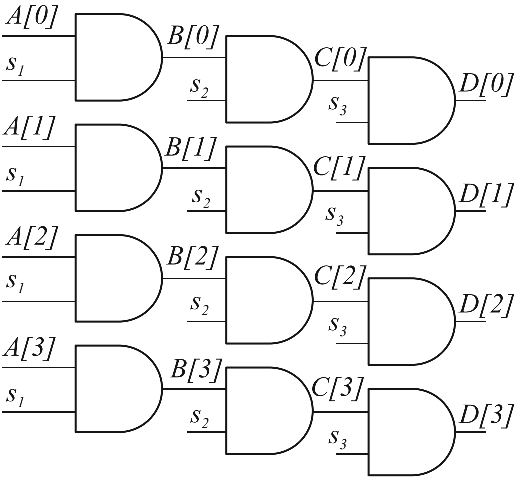
Finally, the function *analyzeBlocks(...)* finds (1) input words on support boundaries which are not directly supported by transparent signals, and (2) output boundaries where words are not fully feeding into transparent signals.

4.3 Running Examples

The following three examples demonstrate how the proposed algorithm works on various cases.

Example 1 Figure 5.5 shows a compound word composed of three depth-one transparent words. First, high-fanout signals, s_1, s_2, and s_3, are collected as control candidates. Then depth-one transparent words, $s_1 \rightarrow (B \equiv A)$, $s_2 \rightarrow (C \equiv B)$ and $s_3 \rightarrow (D \equiv C)$ are proved by Lines 3–5 in Fig. 5.4. Lines 7–14 are skipped because all words found above are either fully connected to another word or support nothing. The function *disposeWords(...)* might drop some depth-one words in this

Fig. 5.5 A compound word composed of four depth-one words



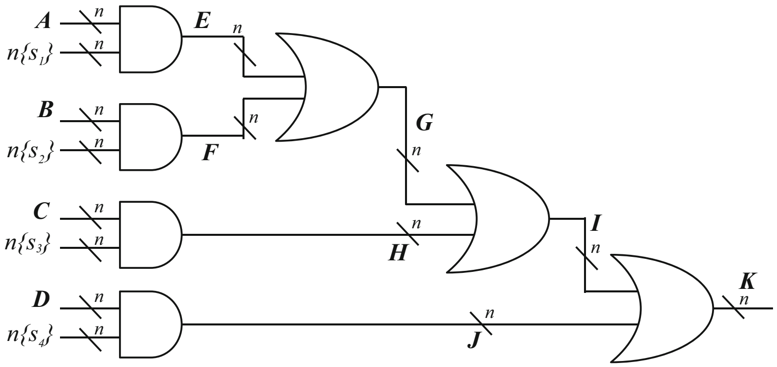


Fig. 5.6 An example with proceeding words

circuit when certain strategies are applied. See Sect. 5.2 for a further discussion. If no word gets dropped, the whole circuit is reported as one transparent logic block, where A and D are input and output boundaries, respectively.

Example 2 Figure 5.6 demonstrates how proceeding words can be found by the proposed algorithm. Similarly, s_1, s_2, s_3, and s_4 are recognized as control candidates. Then $s_1 \rightarrow (E \equiv A)$, $s_2 \rightarrow (F \equiv B)$, $s_3 \rightarrow (H \equiv C)$ and $s_4 \rightarrow (J \equiv D)$ are proved as depth-one transparent words.

Starting from E, *findPossibleCombinations(...)* finds that the combination of E, F, and G satisfies the definition of a proceeding word, so $\{s_1, s_2\}$ is a new combination of controls. According to this control set, the function *findTransparency(...)* returns $s_1 s_2' \rightarrow (G \equiv A)$ and $s_1' s_2 \rightarrow (G \equiv B)$. Following the similar procedure, I is proved as transparent from A $(s_1 s_2' s_3')$, B $(s_1' s_2 s_3')$ and C $(s_1' s_2' s_3)$. Finally I and J are associated together and K is transparent from A $(s_1 s_2' s_3' s_4')$, B $(s_1' s_2 s_3' s_4')$, C $(s_1' s_2' s_3 s_4')$, and D $(s_1' s_2' s_3' s_4)$. The only word on the output boundary, K, is a depth-four transparent word supported by A, B, C, and D.

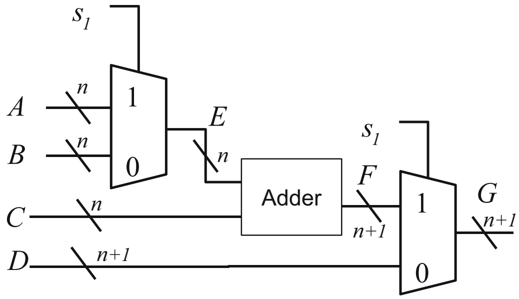


Fig. 5.7 An example with disjoint transparent blocks

Example 3 Consider Fig. 5.7 as the input circuit. First, s_1 is recognized as a high-fanout signal. Then *applyMinterm(...)* in *findTransparency(...)* (Fig. 5.3) finds $s_1 \rightarrow (\{E, G\} \equiv \{A, F\})$ and $s_1' \rightarrow (\{E, G\} \equiv \{B, D\})$. Then *analyzeWords(...)* in Fig. 5.7 merges the two sets of paths into one word, $\{E, G\}$, and then partitions the word into $(E \equiv s_1 A + s_1' B)$ and $(G \equiv s_1 F + s_1' D)$, because sources of E (bits of A) are primary inputs, while those of G (bits of F) are internal signals. There is no more transparent word in this circuit.

There are two disjoint transparent blocks, because all paths through the adder are not transparent. For the block with output E, A and B are reported as support words, while F and D are support words for the other block with G as the output.

5 Practical Challenges

Finding transparent words and performing perfect reverse engineering for real circuits can be challenging, but the proposed algorithm provides more possibilities to address issues that cannot be resolved by structural approaches.

5.1 Generalized Transparent Words

Often a word is transparent from words with different bitwidths—for example,

$$C[m-1:0] = s?A[m-2:0] : B[m-1:0]. \tag{5.9}$$

The most significant bit of C, $C[m-1] = s'B[m-1]$, is not NPN isomorphic to other bits of C. Hence this word is partitioned into one word with $m-1$ bits and one with 1 bit by the proposed algorithm.

Also, it is possible that control signals can be part of input data words, i.e.

$$C[m-1:0] = s?A[m-1:0] : B[m-1:0], \tag{5.10}$$

where $s \equiv A[m-1]$. That is, $C[m-1] = sA[m-1] + s'B[m-1] = A[m-1] + B[m-1]$. Similarly, $C[m-1]$ is different from other bits of C. These excluded bits are discarded by *disposeWords(…)* because of bit-width differences, and then the transparent boundaries are imperfect.

The above cases can be handled by modifying *analyzeWords(…)* in Fig. 5.3. For both the above cases, when s is 1, $C[m-1]$ is constant, which is excluded from **Candidates**. When s is 0, $C[m-1]$ is transparent from $B[m-1]$, through a path with depth one, while other bits of C are through paths of depth two. Hence, *analyzeWords(…)* can merge all bits of C into one word even though some bits are controlled by only one minterm, and their depths are different. In other words, we can relax the requirement of NPN isomorphism to achieve *generalized transparent words*.

However, it is possible that the bits with different depths are indeed different words, so a practical reverse engineering process should consider both cases and use other information to revise the word boundary.

5.2 Ambiguity of Transparency

Some data signals are recognized as control candidates because they drive more than three fanouts. For example, a word-level multiplier can be synthesized as a set of adders among internal words,

$$\begin{aligned} C[2m-1:0] &= A[m-1:0] \times B[m-1:0] \\ &= A[m-1:0]B[0] + A[m-1:0]B[1] << 1 \\ &\quad + \cdots + A[m-1:0]B[m-1] << (m-1). \end{aligned} \tag{5.11}$$

These words are depth-one transparent words from one input word of the multiplier, controlled by bits of the other input word. These depth-one transparent words can be recognized and excluded by *disposeWords(…)*, considering they are directly feeding into non-transparent paths.

However, it is excessive to discard all depth-one transparent words which do not support other transparent words. Consider the example in Fig. 5.5, bits of D are sinks of transparent paths, while they do not support other transparent words and can be discarded.

Also, if a word (a set of MUXes) is switching between one constant word and one variable, these MUXes can be synthesized as AND or OR gates, which comprise depth-one transparent words.

Moreover, compared to the flow in Fig. 5.4, the resulting words and boundaries are different when the function *disposeWords(…)* (discarding some depth-one words) executes before *partitionWords(…)*.

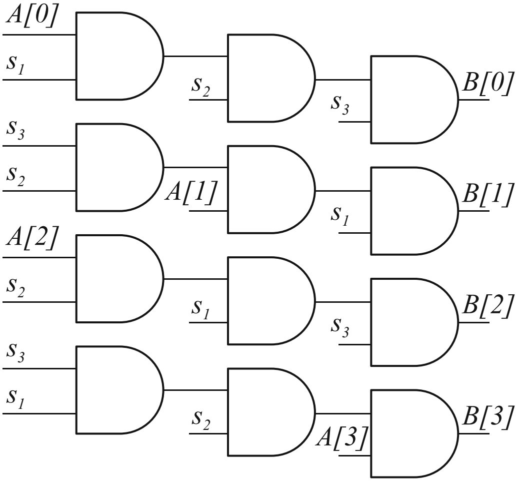


Fig. 5.8 A transparent block which is not resulted by compositions of NPN isomorphism classes

To resolve this issue for reverse engineering, it is preferable to run the proposed algorithm with different settings of *disposeWords(. . .)*, and combine the information of recognized operators to decide suitable boundaries.

5.3 Limitations of Proposed Algorithms

The proposed algorithm only considers composition and propagation of transparency, so it cannot recognize transparent blocks outside these categories.

Consider the circuit in Fig. 5.8. It is a transparent block because $s_1 s_2 s_3 \rightarrow (B \equiv A)$, but it cannot be found by the proposed algorithm. The proposed algorithm will find several depth-one transparent words controlled by s_1, s_2, and s_3, but then signal dependencies cannot satisfy the definition of proceeding words.

Note that this logic block still satisfies the definition of a transparent word, so it can be recognized by finding NPN isomorphism functions and permuting supports. However, searching for all NPN isomorphism classes can be time-consuming, especially if finding and revising ideal support and output boundaries is done.

In real benchmarks, these types of cases do not seem to happen a lot, so it can be omitted by the proposed algorithm without much loss.

6 Experimental Results

The proposed algorithms were implemented in ABC [2]. All experiments were performed on a 16-core 2.60 GHz Intel(R) Xeon(R) CPU with no time limit. All cases were processed as AIGs. Sequential circuits were converted into combinational designs by replacing flip-flops inputs and outputs with primary outputs and inputs, respectively.

As a reference for the functional approach, we implemented a purely structural approach: (1) structural matching is used to locate all 2-to-1 MUXes in the AIGs, (2) signals with the same control are grouped into one word, and these connected words are collected into larger transparent blocks, and (3) words are partitioned into sub-words if they are supported by different input words or drive different output words.

We wanted to compare the efficiency and effectiveness of the structural algorithms versus our functional algorithms applied to highly transparent cases. To select these, the proposed functional algorithm was applied to all 230 cases of the single-output track in the Hardware Model Checking Competition 2014 [3] after their conversion to combinational circuits. For each case, some POs were proved conditionally equivalent to some primary inputs. We computed the proportion of those POs to all POs and ran experiments on the top 10 cases with the highest percentages of transparent POs. Among the 230 cases, there are 20 cases with more than 50% transparent POs, while another 38 cases have more than 25% transparent POs. Table 5.1 shows the statistics of the selected cases after they were converted to combinational circuits. The last column of Table 5.1 indicates the percentages of transparent POs over all POs. The *6sxxx* cases are industrial problems from IBM and the *beem* examples come from different applications areas like protocols, planning, scheduling, communication, or puzzles.

Table 5.1 Statistics of the selected benchmarks from HWMCC'14 [3]

Case name	PI #	PO #	AND #	Trans. PO %
6s195.aig	1344	1258	8046	87.1
beemfrogs1b1.aig	323	159	8493	86.0
6s171.aig	1357	1263	8074	84.6
beemloyd3b1.aig	237	118	3970	82.1
6s282b01.aig	1977	1934	10,264	81.2
6s384rb024.aig	22,367	14,953	47,933	79.0
6s206rb103.aig	37,847	28,644	103,375	71.4
6s302rb09.aig	36,962	27,777	100,571	70.3
6s348b53.aig	15,797	15,561	89,567	70.1
beemldelec4b1.aig	2559	1215	34,252	67.5

6.1 Comparison Between Functional and Structural Approaches

Table 5.2 shows the comparisons between the structural approach and the proposed method. Column 2 indicates the total number of signals (AIG nodes or primary inputs) that were classified as belonging to words. Columns 3–8 (labeled *Structural Approach*) give the statistics of the transparencies found using the reference structural approach. Column 3 lists the total number of structural MUXes recognized. Column 4 lists the number of AIG nodes plus inputs covered by all the transparent logic blocks found; Column 5 gives the (minimum, maximum) widths (the number of MUXes grouped together as a word) of found and partitioned words, and Column 6 shows the (minimum, maximum) depths of transparent words on boundaries. The depth of each word is the total number of AIG nodes between itself and the primary inputs, where one MUX is counted as depth 2. Column 7 (labeled *Forward*) lists the total number of signals which are in the transparent block where all input words are primary inputs. Column 8 shows the run-time of the overall structural approach. Here we only identify 2-to-1 MUXes and MUXes with negation on outputs or inputs. We omit counting words (after partitioning) with less than 4 bits. Columns 9–13 (labeled Proposed Functional Approach) show similar statistics for the proposed algorithm. Here we include all depth-one transparent words when counting signals or depths of transparent blocks.

Observation of Structural Results Table 5.2 shows that most benchmarks contain wide transparent words. The runtimes show that this approach is very efficient as expected.

Although these cases have high percentages of transparent POs, for some cases (beemldelec4b1.aig) the structural approach cannot find any transparent words reachable from primary inputs. Many MUXes are recognized but there are several reasons why the structural approach misses many transparent words:

1. Structural matching only considers standard 2-to-1 multiplexers, while there are other types of transparent functions.
2. Many of the identified MUXes are controlled by different selection signals, and thus lead to words of less than 4 bits, which are excluded in the analysis. Moreover, some words are partitioned into small words because their output or input dependencies are different.
3. Forward transparent words are required to be reachable from primary inputs through fully transparent paths. If a transparent word originates from the output word of an arithmetic operator (e.g., words G and F in Fig. 5.7) or a depth-one transparent word, it would not be reported, yet many MUXes would be involved in such a transparency.

Although quite fast, this approach itself is not enough for finding many of the whole transparent blocks that exist in these benchmarks as shown in the columns which show the forward and total signals found by the functional approaches.

Table 5.2 Experimental results of the structural and functional approaches on ten selected cases from HWMCC’14 [3]

Case	Total	Structural approach						Proposed functional approach				
name	Sig. #	Mux #	Sig. #	Widths	Depths	Forward	Time (s)	Sig. #	Widths	Depths	Forward	Time (s)
6s195.aig	9390	2357	8090	4, 512	2, 12	4620	0.133	8820	4, 512	1, 16	7221	0.183
beemfrogs1b1.aig	8816	2016	5381	6, 8	2, 32	520	0.142	6365	4, 18	1, 34	827	0.267
6s171.aig	9431	2362	8135	4, 512	2, 12	4737	0.161	8847	4, 512	1, 16	7249	0.246
beemloyd3b1.aig	4207	985	2771	6, 8	2, 28	360	0.125	3071	4, 13	1, 29	1512	0.167
6s282b01.aig	12,241	2472	8490	4, 982	2, 54	6175	0.142	10,369	4, 982	1, 66	8072	0.291
6s384rb024.aig	70,300	14,492	53,380	4, 3723	2.8	46,083	0.183	57,717	4, 3663	1, 11	51,918	0.591
6s206rb103.aig	141,222	28,684	106,822	4, 737	2, 8	87,328	0.258	116,996	4, 651	1, 10	101,388	2.365
6s302rb09.aig	137,533	27,818	103,864	4, 590	2, 8	84,943	0.250	113,803	4, 471	1, 10	98,888	2.250
6s348b53.aig	105,364	28,775	66,356	4, 427	2, 16	52,107	0.217	78,757	4, 427	1, 17	69,525	1.274
beemldelec4b1.aig	36,811	8458	14,561	5, 41	2, 76	0	0.167	24,169	4, 9	1, 104	9199	1.857

Comparing Functional and Structural Approaches Based on Table 5.2, we observe the following:

1. According to the signals covered by transparent blocks, the proposed functional approach can find more and larger transparent blocks. Note that for some cases the differences are not huge, which leads to a conclusion that most transparent logic in those cases are comprised by standard 2-to-1 MUXes.
2. The minimum and maximum widths of transparent words are different for the structural and functional approaches.
3. For most cases, the proposed functional approach can find deeper transparent paths, because the functional approach can find depth-one transparent words and compose them into larger transparent blocks.
4. In general, the proposed algorithm can find many more transparent words reachable from primary inputs. It might be that some transparent paths start with MUXes between a constant integer and an input word, which cannot be recognized by the structural approach. Therefore all transparent words supported by that will not be reported as forward transparent words by the structural approach.
5. Although the functional method takes more time than the structural approach, the run times for the selected cases do not exceed 3 s. Hence the proposed algorithm is efficient enough for many applications.

For real applications, the particular final usage of the found words will dictate a suitable balance between performance and the number of proved words.

Even though some transparent words found by the functional approach may be discarded when combined with other techniques for recognizing arithmetic operators, it is better to have more candidates words for more refined reverse engineering applications.

6.2 Experiments on Unrolled Circuits

Table 5.3 shows the experimental results of running the proposed algorithm on circuits unrolled for two and three time frames. The columns are similar to those in Table 5.2.

Observation of Unrolled Circuits Table 5.2 shows that, for each circuit, the number of signals covered by transparent blocks grows as the number of time frames increases. The changes of other statistics vary among the different circuits:

1. The maximal width of words remains the same for most of the cases, but for some, the maximal width decreases after unrolling. The reason is, after unrolling, more words are partitioned into smaller words because some bits support different words in their fanout cones.
2. For many cases, the maximal depths of transparent blocks increase as the numbers of time frames increases. These deeper paths indicate that some transparent paths continue from the first time frame to the second, and some

Table 5.3 Experimental results of the functional approaches on unrolled cases from HWMCC’14 [3]

Case	Two timeframes					Three timeframes				
name	Sig. #	Widths	Depths	Forward	Time (s)	Sig. #	Widths	Depths	Forward	Time (s)
6s195.aig	16,149	4, 512	1, 18	9693	0.300	23,484	4, 512	1, 20	10,112	0.533
beemfrogs1b1.aig	12,750	4, 18	1, 41	1674	0.591	19,135	4, 18	1, 41	2521	0.874
6s171.aig	16,194	4, 512	1, 18	9740	0.358	23,547	4, 512	1, 20	10,178	0.533
beemloyd3b1.aig	6195	4, 13	2, 29	3055	0.217	9319	4, 13	1, 29	4598	0.308
6s282b01.aig	18,782	4, 881	1, 68	9722	0.500	25,332	4, 881	1, 72	11,584	1.074
6s384rb024.aig	101,059	4, 2993	1, 13	85,193	1.915	144,299	4, 2691	1, 17	115,368	4.205
6s206rb103.aig	207,172	4, 585	1, 12	164,619	6.553	297,312	4, 585	1, 19	219,035	13.580
6s302rb09.aig	201,639	4, 455	1, 12	160,876	6.303	289,451	4, 455	1, 19	214,364	12.931
6s348b53.aig	141,452	4, 256	1, 21	117,946	3.789	203,975	4, 256	1, 31	160,923	7.547
beemldelec4b1.aig	48,308	4, 9	1, 116	18,368	4.438	72,447	4, 9	1, 116	27,537	7.012

continue into the third time frame. If the maximal path is not connected to another transparent path in the next time frame, the maximal depth remains the same.

3. The total number of signals in transparent blocks supported by primary inputs (labeled Forward) increases for all circuits, but the growth rate is distinct for each circuit. For some cases, many transparent POs are connected to other transparent paths in the next time frame, so after unrolling, there are more internal transparent words reachable from primary inputs.

The distinct statistics of transparent blocks found in different circuits show that finding transparent logic may be important for understanding circuit properties, and by finding as many as possible transparent blocks the proposed approach may be very useful in providing useful information about a circuit.

7 Conclusions

This paper presented algorithms for finding transparent logic, which can be used to highlight word-level information in gate-level circuits. A functional approach was proposed to identify transparent logic in combinational circuits. Some challenges for finding the most accurate boundaries for the transparencies were discussed. Experimental results demonstrated that the proposed algorithms can be very effective in extracting words as well as some control logic.

Future work will include integrating the proposed method with other reverse engineering techniques that can identify word-level operators. Through iterations between different approaches, word-level information and found boundaries can be revised benefiting both finding more transparencies and identifying operators. A final goal would be a framework for fully reverse engineering gate-level designs.

Acknowledgements This work is supported in part by SRC contract 2265.001 and by the NSA via the TRUST grant. We also thank industrial sponsors of BVSRC, Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific for their continued support.

References

1. S. Awodey, *Category Theory* (Clarendon Press, Oxford University Press, Oxford, New York, 2006)
2. R. Brayton, A. Mishchenko, Abc: an academic industrial-strength verification tool, in *Computer Aided Verification* (Springer, Berlin, 2010), pp. 24–40
3. G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, K. Heljanko, Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. J. Satisfiability Boolean Model. Comput. **9**, 135–172 (2016)
4. Y.Y. Dai, K.Y. Khoo, R. Brayton, Sequential equivalence checking of clock-gated circuits, in *Design Automation Conference* (ACM, New York, 2015)

5. M.C. Hansen, H. Yalcin, J.P. Hayes, Unveiling the iscas-85 benchmarks: a case study in reverse engineering. IEEE Des. Test Comput. **16**(3), 72–80 (1999)
6. W. Li, Formal methods for reverse engineering gate-level netlists. Master's thesis, University of California, Berkeley (2013)
7. W. Li, G. Adria, P. Subramanyan, W.Y. Tan, A. Tiwari, S. Malik, N. Shankar, S. Seshia, Wordrev: finding word-level structures in a sea of bit-level gates, in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (IEEE, New York, 2013), pp. 67–74
8. M. Soeken, B. Sterin, R. Drechsler, and R. Brayton, "Simulation graphs for reverse engineering," in *FMCAD*, (2015), pp. 152–159.
9. M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R.K. Brayton, G.D. Micheli, Heuristic NPN classification for large functions using AIGs and LEXSAT, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)* (2016)
10. P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W.Y. Tan, A. Tiwari, N. Shankar, S. Seshia, S. Malik, Reverse engineering digital circuits using structural and functional analyses. IEEE Trans. Emerg. Top. Comput. **2**(1), 63–80 (2014)

Chapter 6
Automated Pipeline Transformations with Fluid Pipelines

Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau

1 Introduction

In digital design, cycle time and pipeline depth are set early due to their impact on other design parameters. It takes numerous long iterations between design and implementation, to meet a desired cycle time, making it challenging and costly to meet the original specifications. Existing EDA tools allow automatic optimizations such as gate sizing, retiming, and time borrowing, but for a synchronous system, such transformations preserve cycle accuracy. Elastic Systems [8, 9, 21], an alternative to the fixed pipeline paradigm, are based on the assumption that system correctness does not depend on latency (number of clock cycles) between two events, but only on their order [4, 23]. Such paradigm allows for the insertion of new stages later in the design time, when physical implications of micro-architectural choices are known and the circuit timing characteristics are well understood, without breaking the circuit correctness [4], thus improving the ability to meet timing requirements.

Although inserting additional pipeline stages [2, 18] is possible in Elastic Systems, this insertion is constrained by the presence of sequential loops,[1] which significantly reduces its applicability because complex circuits such as processors include loops. Sequential loops are of interest because early approaches for Elastic Systems always maintain the completion order of operation, due to the automated

[1] Cycles in the graph representing the connections between registers, not to be confused with program loops.

R.T. Possignolo (✉) • E. Ebrahimi • H. Skinner • J. Renau
Department of Computer Engineering, University of California, Santa Cruz, Santa Cruz, CA 95064, USA
e-mail: rpossign@ucsc.edu; eebrahim@ucsc.edu; hskinner@ucsc.edu; renau@ucsc.edu

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_6

flow used to transform synchronous circuits into elastic. Thus, the addition of extra pipeline stages inside sequential loops has been shown to degrade the overall system throughput [4, 16]. Throughput losses can be mitigated [2] but the whole system remains constrained by the throughput of the worst sequential loop, even when that loop is not used.

In contrast, Out-of-Order (OoO) execution is omnipresent in modern digital design and improves system throughput. Recently, Fluid Pipelines, a framework that integrates OoO execution into Elastic Systems has been proposed [21, 22]. Fluid Pipelines enable unordered completion, by relying on designer annotations in the code where ordering can be changed. Fluid Pipelines are a generalization of Elastic Systems, since without user annotations, they behave like Elastic Systems. User defined elasticity [3] is thought to improve design methodologies [23] since it reduces the pressure on timing constraints and let logic designers focus on functionality rather than physical implementation.

Fluid Pipelines reclaim the throughput losses from the automated conversion [21] that is typical in Elastic Systems. The automated flow of Elastic Systems transforms a sequential circuit to an elastic one by inserting special control operators: Fork and Join. In short, Fork is used when the output of one stage forks to multiple stages, whereas Join is used when parallel data paths reunite, therefore, the inputs of a stage come from separate stages. The Join operator requires all the inputs to be valid in order to proceed, e.g., the inputs to an adder unit need to be ready at the same time for the operation to take place. Fluid Pipelines rely on Branch and Merge operators [11, 14] to implement the Out-of-Order behavior. They are dual to Fork and Join, but differ in behavior from them. When there is no dependency between the inputs of a block, a Merge operation is said to take place. Merge differs from Join because it is triggered when at least one of the inputs is valid (i.e., it has "or-causality"). In addition, only data from one of the inputs is consumed at each cycle. Its dual, Branch, propagates data to only one of multiple output paths, as opposed to sending data to all of them. This behavior is found in many digital designs, like a Floating Point Unit (FPU) with independent operations; or a network router, where packages come from different inputs and propagate to a single output.

To evaluate Fluid Pipelines performance, a designer or a tool needs to estimate the throughput of a given pipeline configuration. A methodology based on Coloured Petri Nets (CPN) [15] can be used to that end [22]. This methodology allows a designer to quickly explore the design space without performing slow RTL or gate-level simulation of every design point. In some cases it may be hard to accurately model the system as a CPN, and thus it may be more suitable to perform cycle accurate simulation to determine the system performance.

Experimental results show that for an OoO core, Fluid Pipelines improve the optimal energy-delay (ED) point by increasing performance by 17% and reducing energy by 13%, when compared to previous Elastic Systems. A simpler FPU benchmark shows even better results, with improvements of up to 176% in performance, and 5% less power consumption. By using CPN models, it is possible to explore the Pareto frontier and select different interesting design points, depending on a specific application.

The remainder of this chapter is organized as follows. In Sect. 2, we describe other approaches that try to improve the ability of a designer to meet timing specifications in digital design. Then, Sect. 3 describes Fluid Pipelines, with its semantics, constraints, and possible pitfalls. In Sect. 4, we show how CPNs can be used to model systems based on Fluid Pipelines. Finally, Sect. 5 provides an experimental evaluation of Fluid Pipelines, comparing with previous Elastic System approaches. We wrap up in Sect. 6, commenting on research challenges and future directions for both system and EDA tool designers.

2 Related Work

Software Dataflow Networks [1] uses OoO and Speculation in parallel software scheduling. By speculating whether dependencies in the code being executed are true dependencies, the flow can improve execution speed. In case of mis-speculation, execution is re-triggered, similarly to what occurs in the case of branch mis-prediction. Some of the concepts used in Software Dataflow Networks are similar to the ones used by Fluid Pipelines, but in this context, they are applied in a higher abstraction level (macro-architecture) to improve parallel execution. Fluid Pipelines also avoids speculation by giving control to the designer.

High Level Synthesis (HLS) [20] is a technique that uses high-level programming languages to generate hardware. By avoiding describing hardware directly, HLS allows designers to focus on functionality, while the HLS tools take care of timing and pipelining during scheduling [6]. Traditionally, HLS generates synchronous circuits (i.e., not elastic), and thus scheduling is limited by the presence of dependency loops. HLS could leverage Fluid Pipelines to enable changing the number of stages in such loops, and are orthogonal in that regard. In fact, this could improve HLS design time by avoiding multiple iterations to meet timing (i.e., by adding flops without going back to the RTL description).

Dimitrakopoulos et al. [11] explore the reduction of buffering to support multi-threading in Elastic Systems. Their work presents a certain amount of Out-of-Ordering on an inter-thread basis (i.e., no ordering enforced between different threads). Fluid Pipelines allow full Out-of-Order execution. The analogy would be that of a Simultaneous Multithread (SMT) in-order core versus an Out-of-Order core. Also note that this work brings concepts of threads to circuit level decisions, which is usually not performed in digital design.

Elastic Coarse Grain Reconfigurable Arrays (CGRAs) [14] are an approach for coarse grain reconfigurable logic that relies on elastic interfaces for flow control. Elastic CGRAs use Branch and Merge operators across basic blocks (connecting inputs and outputs from different accelerator units), while Fork and Join are used within basic blocks (in the calculation itself). This is conceptually similar to Fluid Pipelines, but limits where each operator can be used. Elastic CGRAs are also based on an automated flow that can differentiate to some extent between ordered and unordered operators. Such a flow could be further extended to be used with Fluid Pipelines, but would most likely require more information from the designer that what is currently done in RTL code.

To mitigate throughput loss in Elastic Systems, different approaches have been proposed. The Eager Fork operator [8] lets one of the paths start executing even when the parallel path is not ready, whereas FIFOs allow for more buffering [23]. Early Evaluation [2] determines which inputs in merging paths are actually needed (such as in a mux), and only waits for those inputs. The next input from other paths is ignored to maintain correctness. Nevertheless, those approaches do not change system semantics. This becomes problematic when one of the paths takes multiple cycles to complete. Then, back pressure propagates to the preceding stages. Fluid Pipelines avoid this scenario by not waiting for parallel paths unless it is needed.

3 Fluid Pipelines

Elasticity is defined as functional correctness depending only on the order of events and not the exact arrival time or clock cycle [5]. Events, also called tokens, are meaningful data, from a designer perspective, flowing through a channel. A typical execution example of a circuit implementing the elastic property is shown in Fig. 6.1, where the arrival of a valid token is represented by a number in a given cell. When a result is produced, the token is consumed and can no longer be used. Empty cells in the table denote that no new data has arrived in that cycle. Note that the latency between events is arbitrary.[2]

To implement this behavior, Elastic System approaches have traditionally relied on a pair of handshake signals: *Valid* (V) and *Stop* (S),[3] which determine three states: *transfer* ($V = 1, S = 0$), *idle* ($V = 0$), and *retry* ($V = 1, S = 1$) [8]. Fluid Pipelines keep this convention, but could be built using other equivalent approaches. The name Fluid Channel is used to denote a data bus and its associated control

Clock Cycle	1	2	3	4	5	6	7	8
A	0	4				3		
B	1		2	3				
A+B		1		6				6

Fig. 6.1 The figure depicts the functionality of an elastic adder. Operands A and B may arrive at different clock cycles and the latency may be arbitrary. Elastic Systems functionality does not depend on the exact cycle events happen, but rather on their order

[2] In practice, a circuit implementing elasticity will most likely be deterministic depending on the input set, but this is not a formal requirement of the Elastic Systems specification.

[3] Other equivalent naming conventions have been used, e.g., Elasticity has been expressed in terms of FIFO operation [23].

signals. Towards this chapter, a Fluid Channel will be often represented as a single arrow for the sake of cleanness.

3.1 Communication and Flow Control

The inter-stage communication is performed through the help of *Elastic Buffers* (EBs), storage units that replace registers, which include handshake signals both on the input and output interface. Figure 6.2 shows the interface of an EB with input and output control signals. Multiple implementations of EBs have been proposed in the literature (see [10] for some). In this chapter, we do not discuss the trade-offs involved in each implementation, and the experimental evaluation uses the implementation presented in Fig. 6.3 that has buffering capacity of 2.

In general, each stage can have multiple input/output channels. To support this, Fluid Pipelines rely on two pairs of control operators: one that maintains the ordering (Fork and Join) and one that does not guarantee ordering (Branch and Merge). The basic implementation of these operators is depicted in Fig. 6.4. In

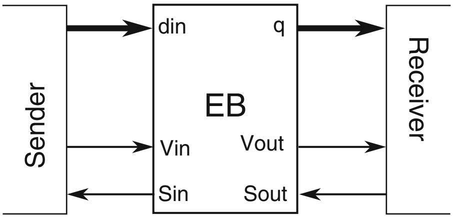
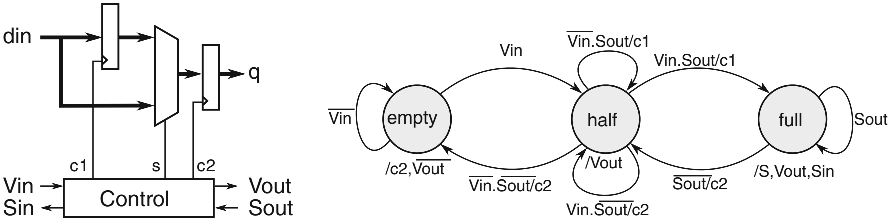


Fig. 6.2 An elastic buffer is shown with respective sender and receiver sides, *thick lines* denote multi-bit buses. An elastic buffer contains a data bus (*din* and *q*) and the valid (V_{in}, V_{out}) and stop (S_{in}, S_{out}) handshake signals. Elastic buffers are the basic construct blocks of Elastic Systems and can be viewed as queues with a limited size


din
q
c1
s
c2
Vin
Sin
Control
Vout
Sout
Vin
$\overline{\text{Vin}}$.Sout/c1
Vin.Sout/c1
$\overline{\text{Vin}}$
empty
half
full
Sout
/c2,$\overline{\text{Vout}}$
/Vout
$\overline{\text{Vin}}$.$\overline{\text{Sout}}$/c2
$\overline{\text{Sout}}$/c2
/S,Vout,Sin
Vin.$\overline{\text{Sout}}$/c2

(a) EB implementation

(b) State diagram of Control block

Fig. 6.3 The elastic buffer implementation assumed in this chapter is shown here, *thick lines* denote multi-bit buses. (**a**) Shows the datapath implementation with two registers, and (**b**) shows the state diagram of the control block. This implementation works as a buffer of size 2 with variable latency [8], but multiple implementations of elastic buffers have been proposed [10]

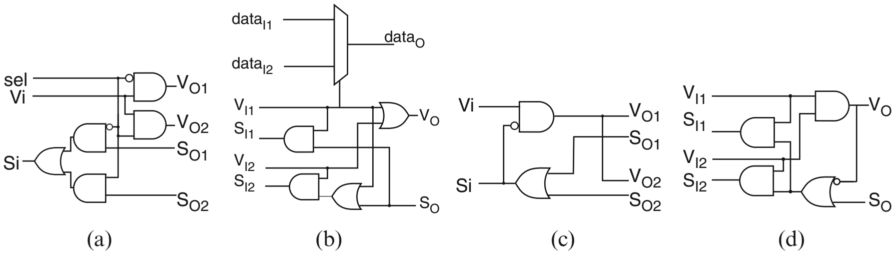


Fig. 6.4 The four operators used by Fluid Pipelines are shown in the figure for the two input or two output versions. (**a**) Shows the Branch operator which propagates data to one out of *n* possible outputs, (**b**) shows the Merge operator that propagates data from any of the *m* inputs, (**c**) shows the Fork operator that propagates data to all the *n* outputs, and (**d**) shows the Join operator that propagates data when all the *m* inputs contain valid data. The operators translate the intended functionality of a circuit and enable better design space exploration. Branch and Merge are used when the relative order of operations can be broken, while Forks and Joins enforce ordering. Note the difference in the handling of "valid" and "stop" signals

Fig. 6.4a, *sel* is a data-dependent selection signal that indicates to which output the data will propagate. The operators can be easily extended to more than two inputs/outputs.

Branch is used when the datapath forks into multiple paths, but data should propagate to only one of them. This is controlled by the selection signal. For instance, an operation in an FPU only needs to propagate to the appropriate functional unit, and the selection signal is encoded by the operation bits. Merge operates as an arbiter: multiple senders compete for a single channel. The sender that wins the arbitration propagates its data. In our FPU example, a Merge would be used at the end of the functional units when results from each unit are collected. Another way to think of the Merge is that it fires when at least one of its inputs contains valid data. This is known as *disjoint or-causality* and introduces the *or-firing* rule to the context of Fluid Pipelines. For simplicity and without loss of generality, the proposed implementation in Fig. 6.4b has simple fixed-priority, but can be replaced with any of the existing elaborated arbitration schemes such as Round-Robin [21].

In general, Branch and Merge cannot be automatically inserted like Fork and Join, because they alter the relative order between events. As a result, the programmer is responsible for inserting them when needed. For example, in a complex Floating Point Unit, just one Branch and Merge pair is needed after the normalization and denormalization stages to indicate that the floating operations can complete out of order. On the other hand, the Fork and Join operators can be automatically inserted in a similar way as the insertions performed in traditional Elastic Systems. Branch and Merge can be performed with direct Verilog/VHDL instantiation or just code annotations. To present, user annotations have been used to determine which operators can be unordered. More automated approaches, like language support, are still open research questions that need to be addressed.

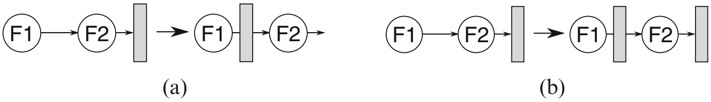


Fig. 6.5 (**a**) Retiming is the operation of moving registers across combinational logic, it is used to balance the pipeline, (**b**) ReCycling is the operation of changing, usually adding, registers to the pipeline. Retiming and ReCycling are used to improve the circuit frequency, but ReCycling decreases the throughput of Elastic Systems when applied to sequential loops

3.2 RePipelining: Optimizing Fluid Pipelines with ReCycling and Retiming

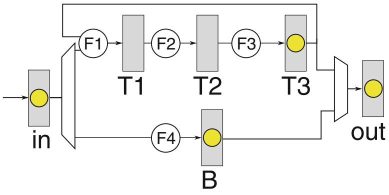
As mentioned, Fluid Pipelines can be optimized by means of pipeline transformations. Modern EDA tools perform operation such as gate sizing, time borrowing, and logic replication to help improve timing and, hopefully, meet design specifications. All those operations preserve cycle accuracy and can be applied to most synchronous circuits, including Fluid Pipelines. The main advantage of Elastic Systems and Fluid Pipelines is the ability to change the number of pipeline stages without breaking the system behavior.[4]

To improve the frequency of Elastic Systems, it is possible to move EBs across circuit blocks (Retiming) [2] (Fig. 6.5a) or to insert additional stages in slower paths (ReCycling) [2] (Fig. 6.5), ReCycling can also remove pipeline stages from non-critical paths for power/area optimization. Retiming preserves the sequential behavior of the circuit [2] and thus it can be applied mostly without penalties.

Inserting pipeline stages can be applied to Fluid Pipelines and prior Elastic System approaches, but in prior approaches this comes with a reduction in throughput in cases where pipeline stages are added to sequential loops. In fact, the throughput of the whole system is limited by the loop with the lowest throughput, due to backpressure, even when this loop is not used. The throughput of a cycle can increase with Early Evaluation depending on how often each event occurs [16], but due to back pressure, there is still a limit on such mitigation. Thus, in prior Elastic System approaches, ReCycling is able to reduce cycle time [12] but may decrease the overall system performance in the case of stage insertion in sequential loops [2, 4, 16].

In Fluid Pipelines, on the other hand, unused paths are isolated from the remainder of the circuit by the use of the unordered operators Branch and Merge. Since only used paths are triggered when Branch and Merge are used, unused low-throughput paths do not "contaminate" the overall system performance. This will become clearer in the next sub-section with a simple execution example.

[4] We note that inserting pipeline stages was proposed in synchronous circuits [12], but breaks the cycle accuracy of the circuit and should be used with care.



(a) Loop Case

ID	Path
I1	Bottom
I2	Top
I3	Bottom
I4	Bottom
I5	Top
I6	Bottom
I7	Bottom

(b) Instructions

Fig. 6.6 Toy case to illustrate the Elastic vs. Fluid approaches. (**a**) The test circuit, where *grey boxes* indicate elastic buffers, *circles* represent combinational logic, and *dots* represent registers with a valid token. (**b**) Shows the instructions executed in this example and which path they are assumed to use

3.2.1 Execution Example

To clarify the practical differences in the formalization between Fluid Pipelines and prior Elastic Systems, let us analyze the sample execution in the example in Fig. 6.6, where circles represent combinational logic, boxes represent EBs, and the dots inside boxes represent the presence of valid data (tokens). The paths are mutually exclusive (each operation either takes the top or the bottom path), and the mux near the output EB chooses the appropriate path. The instructions can take either the bottom path or the top path in Fig. 6.6b. The execution traces for traditional Elastic Systems and Fluid Pipelines are shown in Table 6.1.

The execution order of Fluid Pipelines is altered (Table 6.1), note how in cycle 3, it is possible to move I3 to the bottom path, while the top path is still executing. This re-ordering is a result of the “or-firing” rule and it is done because it was specified by the user, and not changed by the tool. In a processor core, the reordering buffer performs this function, while in network-on-chips, the reordering is usually not performed. Since this requirement is application specific, it is left out of this manuscript. We assume that any reordering needed is performed in the design. In the case where order should be maintained, regular Fork and Join operators must be used, causing the design to behave similarly to a Elastic System.

3.3 Fluid Pipelines Deadlock Avoidance

One possible pitfall in Fluid Pipelines design is the possibility of deadlocks. Since control is given to the designer, special care is needed when designing Fluid Pipelines to avoid deadlock prone situations. Two properties are enough to guarantee that Fluid Pipelines are deadlock free: No-Extraneous Dependencies (NED) and Self-Cleaning (SC) [23]. Those properties can be summarized in the following design directives:

Table 6.1 Sample trace for the toy case in Fig. 6.6 considering both regular Elastic Systems and Fluid Pipelines. Each line denotes a clock cycle and in which stage the instruction is at that cycle. Fluid Pipelines improve throughput compared to Elastic Systems

	Elastic						Fluid					
Cycle	In	T1	T2	T3	B	Out	In	T1	T2	T3	B	Out
0	I1						I1					
1	I2				I1		I2				I1	
2	I3	I2				I1	I3	I2				I1
3	I3		I2				I4		I2		I3	
4	I3			I2			I5			I2	I4	I3
5	I4				I3	I2	I6	I5				I2
6	I5				I4	I3	I6		I5		I6	I4
7	I6	I5				I4	I7			I5	I7	I6
8	I6		I5									I5
9	I6			I5								I7
10	I7				I6	I5						
11					I7	I6						
12						I7						

- **No-extraneous dependencies:** If an output *o* of a module does not depend on an input *i* of that module, then *o* should be produced regardless of the existence of valid data in *i*. Also, the dependency list of *o* should be a subset of the inputs of the module.
- **Self-cleaning:** A circuit is self-cleaning if whenever it has produced *n* tokens in its outputs, it has also consumed *n* tokens from its inputs.

These directives do not restrict which designs are possible, but rather how to implement each design. To make it clearer why those properties are important and how the directives work, let us take the example in Fig. 6.7. The synchronous module described in the figure has a pair of inputs (*a* and *b*) and outputs (*c* and *d*), *c* is a function of *a* and *b*, while the value of *d* depends only on the value of *b*. Now, assume a designer wants to implement that module using Fluid Pipelines.

The most straightforward implementation of the block would follow the behavior described in Fig. 6.8, where “xx_valid” and “xx_stop” denote, respectively, the valid and stop bit for the “xx” bus. In this implementation, the circuit waits until all inputs have valid data, and all outputs can accept new data to perform the operation. This is a violation to the NED directive and can cause deadlocks depending on the context in which the block is used. For instance, in cases where the output *d* is connected as a feedback path to *a*, *d* will only produce output when both *a* and *b* are available.

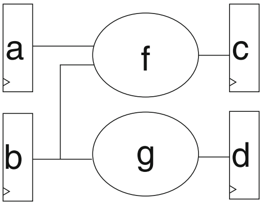


Fig. 6.7 Sample circuit to illustrate deadlock avoidance directives. The circuit contains two inputs (*a* and *b*) and two outputs (*c* and *d*) and performs two operations (*f* and *g*), one of which (*f*) depends on both inputs and the other (*g*) depends only on input *b*. The handshake signals are omitted for the sake of clarity. Fluid Pipelines design uses a few design practices to avoid deadlocks. Those are restrictions on how to implement a given design and not on which designs can be implemented

Fig. 6.8 A pseudo-verilog implementation that generates a deadlock prone implementation of the circuit in Fig. 6.7. This implementation waits until all the inputs have valid data and that all the outputs can receive new data

```
always @ (posedge clk) begin
  if (a_valid && b_valid) begin
    if(!c_stop && !d_stop) begin
      c <= f(a,b);
      d <= g(b);
      c_valid <= true;
      d_valid <= true;
      a_stop <= false;
      b_stop <= false;
    end
  end
end
```

A simple solution to this case is the use of a Fork operator (Fig. 6.9). The Fork operator isolates the handshake handling, and thus avoids the deadlock situation by avoiding the unnecessary wait on a valid signal in *a* to propagate *d*.

The Self-Cleaning property is needed to avoid buffer overflow. Consider the case where a circuit produces *n* inputs per token consumed. If there is a loop where the output of the circuit is connected back to its input, there will be buffer overflow. For a circuit with buffering capacity of *m*, the overflow will occur after *m*/*n* cycles, causing a deadlock.

3.4 Fluid Pipelines Channel Grouping

In high performance design SoCs, it is common to have a guaranteed number of cycles between events. For example, a cache hit in a processor may be known to take three cycles. The issue logic in the processor may start to wake up instructions two cycles ahead. If the design shrinks/increases by one cycle, the time dependence may be broken. These scenarios need to be taken into account in Fluid Pipelines, when adding extra pipeline stages. This information is known by processor architects at design time and can be given to the Fluid Pipelines framework.

```
module fork (in, in_valid, in_stop,
             out1, out1_valid, out1_stop,
             out2, out2_valid, out2_stop);
  // data
  input [N-1:0] in;
  output [N-1:0] out1, out2;

  // handshake
  input in_valid, out1_stop, out2_stop;
  output in_stop, out1_valid, out2_valid;

  if(in_valid && !out1_stop && !out2_stop) begin
    out1 <= in;
    out2 <= in;
    in_stop <= false;
    out1_valid <= true;
    out2_valid <= true;
  end
endmodule

module f_and_g(a, a_valid, a_stop,
               b, b_valid, b_stop,
               c, c_valid, c_stop,
               d, d_valid, d_stop,
               clk);
  // data
  input [N-1:0] a, b;
  output [N-1:0] c, d;
  wire [N-1:0] b1, b2;

  // handshake
  input a_valid, b_valid, c_stop, d_stop;
  output a_stop, b_stop, c_valid, d_valid;
  wire b1_valid, b1_stop, b2_valid, b2_stop;

  input clk;

  fork(b, b_valid, b_stop,
       b1, b1_valid, b1_stop,
       b2, b2_valid, b2_stop);

  always @ (posedge clk) begin
    if (a_valid && b1_valid && !c_stop) begin
      c <= b1;
      c_valid <= true;
      b1_stop <= false;
    end

    if (b2_valid && !d_stop) begin
      d <= b2;
      d_valid <= true;
      b2_stop <= false;
    end
  end
endmodule
```

Fig. 6.9 A pseudo-verilog implementation that solves the deadlock problem by using the fork operator and thus avoiding the extraneous dependency of output *d* on input *a*

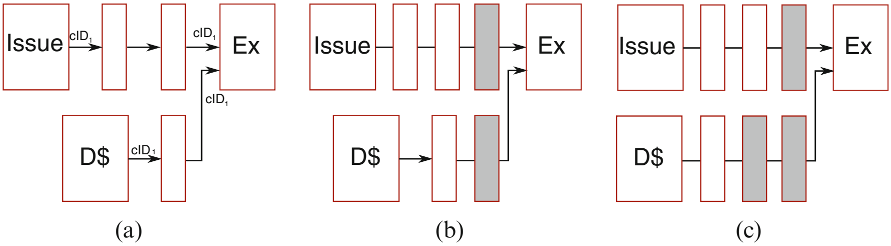


Fig. 6.10 The issue logic of a processor core is shown, *arrows* represent a data/handshake bundle, *shaded boxes* represent elastic buffers and *boxes with names* represents pipeline stages, the channel IDs is denoted with the *arrows*. (**a**) Shows the original logic with two cycles between issue and execute (Ex) units and one cycle between data cache (*D*$) and execute, (**b**) an extra stage (*shaded box*) is added to both channels to create a valid pipeline configuration, (**c**) a different number of stages is added to each path, yielding to an invalid pipeline configuration. By annotating channel IDs, the designer can constrain what pipeline configurations are allowed to guarantee the functional behavior of the circuit, this is specially useful when there is dependency in the latency between two channels

To support this behavior, Fluid Pipelines allow the designer to assign group IDs to a Fluid Channel. For simplicity, channels without user-defined group ID are automatically assigned a unique ID (i.e., empty group). When additional stages are inserted in a channel (or existing stages are removed), all other channels with the same ID get the same amount of extra stages. This guarantees that the relative number of cycles between the channels is kept. There is no requirement that channels share wires or handshake signal and the number of buffers already present in different channels does not need to match [21].

To illustrate this, let us analyze the example of an OoO core. Figure 6.10 shows the instruction wake-up and data cache of an OoO core, the channels connecting wake-up to execute and data cache to execute are assigned the same ID, and thus the same number of stages need to be added/removed to them. A valid solution is shown in Fig. 6.10b, where one extra stage is added (shaded). The circuit in Fig. 6.10c is not a valid solution, since different number of stages is added in each channel.

An implication of channel grouping is that we can always add pipeline stages, but we may not be able to remove pipeline stages in some cases. Figure 6.11 shows two channel groups, out of a fully connected design graph (not represented for cleanness). Group A has the same delay between producers and consumers. This means that any number of stages can be inserted/removed in channels A1 and A2, as long as the number is the same in both channels. Group B has similar constraint, but channel B2 has a number of stages that is larger by one than channel B1. Strictly speaking, this means that pipeline B2 has to have at least one pipeline stage. The minimum number of stages in each case is shown in Fig. 6.11b.

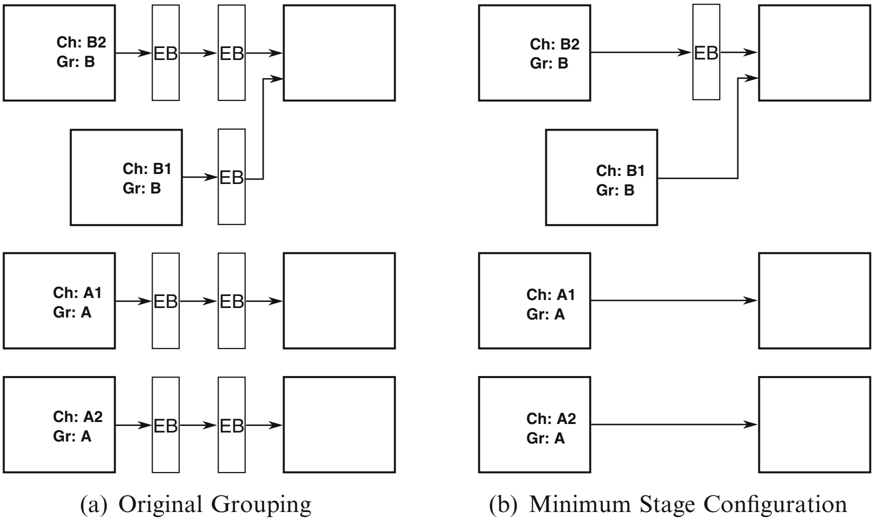


Fig. 6.11 (**a**) Shows two channel groups A and B. Each channel in A has two elastic buffers (EB) between source and sink, whereas the channel $B1$ has one EB and $B2$ has two EBs between source and sink; (**b**) shows the minimum pipeline configuration for both groups, for group B it is not possible to remove all the EBs due to the uneven number of buffers in the original configuration. It is not possible to remove all the stages in the design, but it is always possible to add more stages

3.5 Design Example

Using Fluid Pipelines in dataflow is in general a straightforward task as seen so far. In this section we provide a different example of how memories or Register Files (RF) could be integrated into Fluid Pipelines. When designing Fluid Pipelines it is common to replace registers by EBs, but this is not desirable in the case of RFs, since RFs are supposed to hold a value until a new value is written over it, and after reading from an EB, the data is consumed. Instead, we show here how the memory abstraction (regardless of actual implementation) can be used to model RFs, or any block of memory, in the RTL level.

The idea is to create a wrapper over the memory block that implements the Fluid Pipelines handshaking. We will represent the memory as an array of registers, but a black-box memory, from a memory compiler could be equally used. The Verilog code for the RF is shown in Fig. 6.12.

3.6 Design Overhead

One of the main disadvantages of Fluid Pipelines is the need for design intervention in the RTL code. In this section, we look into how much of what is needed to implement Fluid Pipelines already exists in digital design. In fact, finding points

```
module reg_file(in_data,   in_addr,   in_valid,   in_stop,
                out1_addr, out1_valid, out1_stop,
                out2_addr, out2_valid, out2_stop,
                out1_data, out1_data_valid, out1_data_stop,
                out2_data, out2_data_valid, out2_data_stop,
                clk);
  //addr
  input [M-1:0] in_addr, out1_addr, out2_addr;

  //data
  input  [N-1:0] in_data;
  output [N-1:0] out1_data, out2_data;

  //handshake
  input  in_valid;
  output in_stop;

  input      out1_addr_valid, out2_addr_valid;
  output reg out1_addr_stop,  out2_addr_stop;
  output reg out1_data_valid, out2_data_valid;
  input      out1_data_stop,  out2_data_stop;

  input clk;

  //we always write
  assign in_stop = 0;

  //register array
  reg [N-1:0] registers [REG_COUNT-1:0];

  always @ (posedge clk) begin
    if(out1_addr_valiid && !out1_data_stop) begin
      out1_data <= registers[out1_addr];
      out1_addr_stop  <= 0;
      out1_data_valid <= 1;
    end else if(out1_data_stop) begin
      out1_addr_stop <= 1;
      out1_data_valid <= 0;
    end else
      out1_addr_stop  <= 1;
      out1_data_valid <= 0;
    end

    if(out2_addr_valiid && !out2_data_stop) begin
      out2_data <= registers[out2_addr];
      out2_addr_stop  <= 0;
      out2_data_valid <= 1;
    end else if(out2_data_stop) begin
      out2_addr_stop <= 1;
      out2_data_valid <= 0;
    end else
      out2_addr_stop  <= 1;
      out2_data_valid <= 0;
    end

    if(in_valid) begin
      register[in_addr] <= in_data;
    end
  end
endmodule
```

Fig. 6.12 Sample Fluid Register File using register array in pseudo-Verilog. In this case, it is simpler to keep registers as a memory block, instead of replacing them by EBs, so data is kept after a read operation

where Branch and Merge operators can be inserted is a simple task because most existing designs are inherently elastic.

Elasticity is omnipresent in digital design. Most designs already include signals such as “start,” “done,” “busy,” or “full,” which implement the logic used by Fluid Pipelines. In some cases, like network routers, packages are well defined and routing/contention schemes are already in place. That means that Fluid Pipelines does not require any logic that may be unfamiliar to designers, but only standardizes how to implement this behavior.

To estimate what proportion of existing designs do implement the type of logic required by Fluid Pipelines, we take a look at various designs in OpenCores,[5] an opensource database of digital designs. Even though those designs may not be an ideal representation of practical/commercial designs, it provides a rich estimate from various domains. We counted the number of design implementations that are equivalent (same or inverted signals), partially equivalent (only using one signal or using signals with different meanings), or nonequivalent (not implementing any handshaking) to our handshaking mechanism. We only considered projects marked as “DONE,” in Verilog or VHDL and for which the code is publicly available. Out of 270 projects, 35% are equivalent in most blocks, 10% are equivalent in a few blocks, 20% are partially equivalent (in general, only “start” and “done” signals). 25% implement no or an incompatible handshake. The remaining 10% are IO operations (debouncer, LED control, etc.) or only combinational logic (lookup tables, arithmetic operation, etc.).

These statistics show that the type of handshaking required by Fluid Pipelines is already implemented in a significant number of designs, and therefore, Fluid Pipelines will not introduce design overhead. The designer simply needs to annotate the code. These statistics also show that the type of handshake used by Fluid Pipelines is not new to designers, and implementing them will not be hard for most experienced designers.

4 New Evaluation Methodology

To find the optimal pipeline depth, a designer or tool must estimate the throughput of a pipeline configuration (i.e., number and position of pipeline stages). In theory, this can be accomplished through RTL simulation, cycle-accurate simulators or others. RTL simulations are often slow, especially if a large number of configurations need to be tested. For CPU cores, architects usually rely on standard cycle-accurate simulators, such as ESESC [17]. Still, for other designs it may be hard to write custom simulators. Therefore, a more light-weight methodology can be used to model simple designs faster and evaluate different pipeline configurations early

[5] http://www.opencores.org.

in the design time for space exploration, or late when changes due to physical constraints are included.

A methodology based on Coloured Petri Nets (CPN) [15], a formal framework used to model systems in different areas of computer science, was proposed to evaluate Fluid Pipelines and other Elastic Systems alike [21]. The use of the colored version of Petri Nets is justified by the data-dependent Branch operations that cannot be modeled on the non-colored versions.

CPNs are defined as a bipartite graph of *places* and *transitions*, connected by *arcs*. Places can contain *tokens* that have data value attached to them (*color*). The state of the net (the *marking*) is defined by the number and color of tokens in each place. The initial marking is changed when transitions *fire*. When a transition fires, tokens are subtracted from its input places and added to its output places according to *arc expressions*. There is a *capacity* associated with each place representing the maximum number of tokens in that place, and prevents input transitions from firing.

Definition 1 A **Coloured-Petri Nets** is a tuple CPN $= \langle P, T, A, \Sigma, C, G, E, I, Cap\rangle$:

- P is a finite set of *places*.
- T is a finite set of *transitions*, such that $P \cap T = \varnothing$.
- $A \subseteq (T \times P) \cup (P \times T)$ is a set of directed *arcs*. Let $a.p$ and $a.t$ denote the place and transition connected by a, respectively.
- Σ is a finite set of non-empty *color sets*.
- $C : P \to \Sigma$ is a *color set function* which assigns a color set to each function.
- G is a *guard function* that assigns to each transition $t \in T$ a guard function $G(t) : (\varnothing \cup \Sigma)^{|\bullet t|} \to \{0, 1\}$, where $\bullet t = \{p|(p, t) \in A\}$.
- E is an *arc expression function* that assigns to each arc $a \in A$ an expression $E(a)$, such that the type of $E(a)$ should match $C(a.p)$.
- I is an *initialization function* that assigns to each place $p \in P$ an initialization expression $I(p)$, $I(p)$ must evaluate to $C(p)$.
- $Cap : P \to \mathbb{I}$ is a capacity function that attributes a maximum capacity to each place.

Firing Semantics Let M, a *marking* function, map each place $p \in P$ into a set of tokens $M(p) \in C(p)$. Let $G(t)(M)$ (resp. $E(a)(M)$) denote the evaluation of $G(t)$ (resp. $E(a)$) with the marking M. A transition t is enabled, and said to *fire* when $G(t)(M) = true$ and $\forall a \in \{b|b = (p, t), p \in P, b \in A\}, E(a)(M) <= M(a.p)$, and $\forall p \in t\bullet, M(p) < Cap(p)$, where $t\bullet = \{p|(t, p) \in A\}$. The firing updates the marking function to $M'(p) = (M(p)\ E(p, t) \cup E(t, p) \forall p \in P$.

Timing In order to evaluate digital circuits, one needs to account for timing, which is not included in CPN models. In regular CPNs, only one transaction fires at a given cycle. Without changing the underlying semantics of CPNs, it is possible to change the model so that *every* transition that is enabled at the beginning of the cycle fires. This is a more accurate description of digital circuits and will help determine the number of clock cycles it takes to execute.

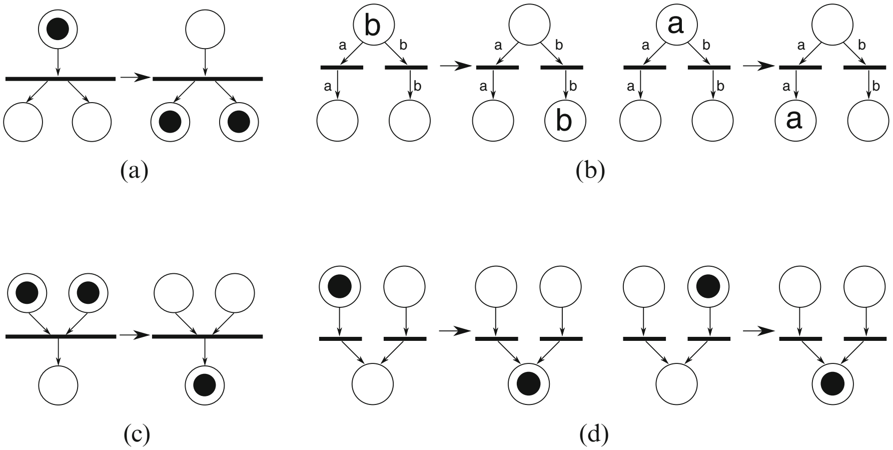


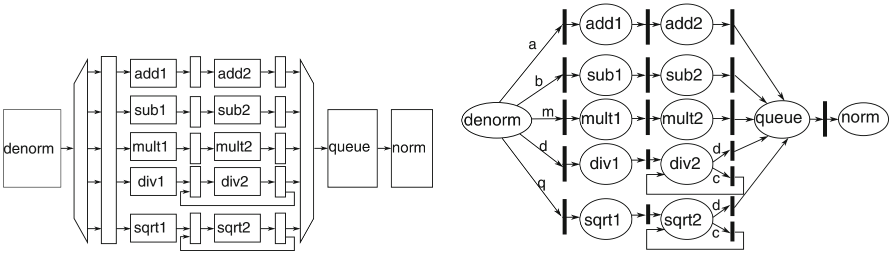
Fig. 6.13 The CPN models of each of the four operators used in Fluid Pipelines. The models are shown before and after firing. Branch is data dependent and thus the arrows are annotated with the expected data. We use CPN models to estimate the overall throughput of Fluid Pipelines and Elastic Systems. (**a**) Fork. (**b**) Branch. (**c**) Join. (**d**) Merge

There is one extra restriction to this formulation. The cardinality of each expression must be 1; this means that for each arc, only one token can be consumed/generated. Also, note that guard functions can only depend on the incoming arcs to a transition. This complies with the constraints defined previously, and thus, avoids deadlocks. The restriction on the cardinality of expressions changes the formalism of CPNs, and a formal analysis of the impact of it needs to be further explored in future work.

Figure 6.13 depicts how the Fluid Pipelines' operators are modeled as CPN transitions. Circles represent places, bars represent transitions, and dots represent tokens in transitions that are not color dependent while letters represent colored tokens. Merge operators do not define priority, and thus, conceptually both transitions can occur at the same time, which is compatible with the theoretical formulation of Fluid Pipelines. While places correspond to elastic buffers, transitions do not have a direct translation from the circuit model. However, they can be mapped from the logic.

5 Evaluation

In this section, we provide some experimental results that show how Fluid Pipelines compare with prior Elastic System approaches, and what kind of trade-offs that Fluid Pipelines enable. We first discuss the evaluation methodology and setup and then show the experimental results.



(a)

(b)

Fig. 6.14 The FPU block diagram (**a**) and the corresponding CPN model (**b**) used to evaluate system performance

5.1 Setup

A fully compliant IEEE-754 in-house FP Unit and a 2-way Out-of-Order FabScalar core [7] are used to evaluate Fluid Pipelines [21]. They were both designed as synchronous (for previous approaches), and annotated with Fluid Pipelines' operators.

A functional block diagram of the FPU unit is presented in Fig. 6.14a, and the CPN model used for the performance evaluation considering Fluid Pipelines is shown in Fig. 6.14b. In this case, the Branch and Merge operators are used. Note how the division and square root modules use the Merge to choose between the loop when the operation is computing or sending the result to the queue when done. Both division and square root take 64 cycles to complete. For regular elastic, the Fork and Join operators are used instead.

The FabScalar-2W OoO core (Fig. 6.15) contains nested loops and interactions between blocks and allows us to explore the scalability of the different approaches. Branch operators are used in the dispatch unit, Exec units, and issue logic. Merge operators are used after the exec units, in the free register pool handling (ROB to Rename path), and in the next program counter calculation (Fetch 1).

Fluid Pipelines are compared against SELF [2, 11] and LI-BDNs [23]. Elastic Systems are implemented with EBs with storage capacity of 2. For LI-BDNs, queues of size 8 were used. In the SELF implementation adding pipeline stages to all the paths that are parallel to the critical path will yield best performance and that is the performance considered in this evaluation.

5.1.1 Benchmarks

For the FPU design, we report maximum and average throughput. Maximum throughput is calculated by using a synthetic workload that only considers the best path (add, subtract, and multiply in this case). The average case is calculated as the throughput over a million random instructions.

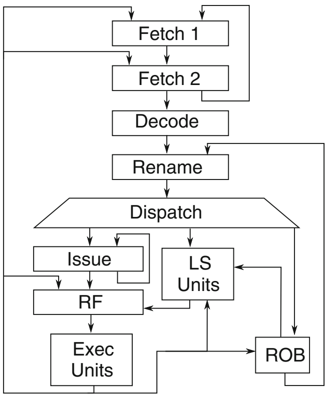


Fig. 6.15 Block diagram of the FabScalar core used to evaluate Fluid Pipelines in this chapter. An OoO core contains a complex structure of nested loops and interactions between blocks. It is used to show the scalability of Fluid Pipelines

For the OoO core, only the average case over the SPEC2006 benchmarks[6] is reported. Per benchmarks results did not add much information and were therefore omitted.

5.1.2 ReCycling

The evaluation considers the addition of extra pipeline stages to each design. Pipeline stages are added to the blocks with the worst delay. Perfect ReCycling/Retiming (perfect balancing of delays) is assumed. Although this is usually not possible, this approximation is sufficient. It is only necessary to ensure that after the insertion of a pipeline stage, the two resulting stages have a delay smaller than the second most critical path before insertion. Also, to account for register overhead, 2FO4 (fan-out-of-4) delay was added per added stage.

To find the most critical pipeline stages, synthesis results for the FPU and previously published data from FabScalar [7] that reports pipeline stage breakdowns were used. The minimum pipeline configuration is the same as in the original non-elastic baseline: 6 for FPU and 13 for the core.

Since ReCycling changes both throughput in instructions per cycle (IPC) and timing, the performance metric used is *throughput*×*frequency* (equivalent to instruction per seconds, IPS). Also, it has been shown that unless power is considered, the ideal pipeline for a design is extremely deep [13]. Therefore, ED is used. Power is estimated from synthesis results for the FPU and ESESC [17] simulations (based on McPAT [19]) for the core. Logic energy consumption (both dynamic and leakage) is assumed to remain roughly constant independent of the number of pipeline stages. However, the dynamic clock energy consumption increases linearly with both frequency and number of registers, and the leakage clock energy increases linearly with the number of registers. This evaluation does not consider the effects

[6] Only the benchmarks that do not require Fortran were used.

of Retiming, that may increase the number of registers added, and assume that the added stages have roughly the same number of flops as existing ones, which may not always be true in the case a stage is added in the middle of an operation.

5.2 *Results*

We first show the design space exploration of the different approaches. In particular, we show that Fluid Pipelines are able to push the Pareto frontier towards better performance and energy efficiency (Sect. 5.3). Then, we report the more detailed results, such as the maximum frequency, throughput, and ED for different pipeline configurations for both the FPU (Sect. 5.4) and Out-of-Order core (Sect. 5.5).

5.3 *Overall Results*

Fluid Pipelines push the design space towards more energy efficiency and better performance. This is accomplished by avoiding false dependencies between concurrent paths. For most of the design points in the design space, Fluid Pipelines improve both better performance and energy. In comparison, LI-BDNs reach better performance than SELF, but at the cost of more energy (and area, not evaluated here).

The Pareto frontier (Fig. 6.16) shows that for OoO core, Fluid Pipelines (FP) deliver both less energy and more performance than SELF. Also, Fluid Pipelines improve the best performance (by 6%, but with 28% less energy) and the best energy point (by 14%, but with 16% more performance). Each point represents a different pipeline configuration, where deeper pipelines tend to improve performance while consuming more energy. In this case, LI-BDN was not used, as it will be explained in the detailed evaluation.

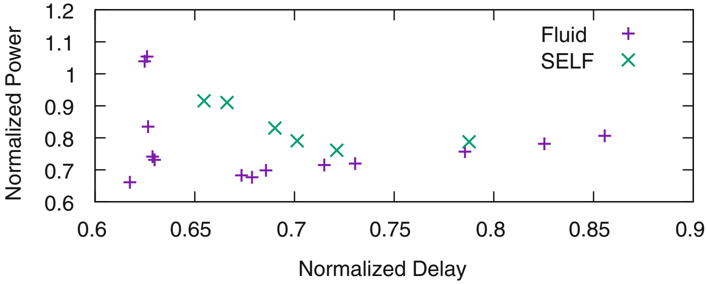


Fig. 6.16 Energy-delay curve for Fluid Pipelines and SELF for the OoO core. *Each point* represents a different number of pipeline stages. The results show that Fluid Pipelines push the Pareto frontier for the OoO core by improving both performance and energy

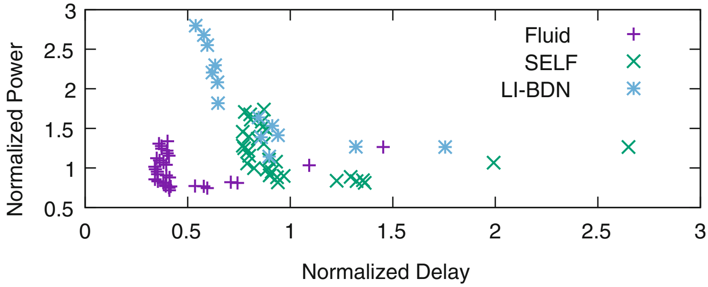


Fig. 6.17 Energy-delay curve for Fluid Pipelines, SELF and LI-BDN for the FPU design. *Each point* represents a different number of pipeline stages. The results show that Fluid Pipelines push the Pareto frontier for the FPU by improving both performance and energy

For the FPU (Fig. 6.17), LI-BDNs result in increased energy consumption due to the increased storage, but improved the performance, when compared to SELF. Fluid Pipelines present the best performance and energy out of the three schemes, since they do not require extra storage. Compared to SELF, Fluid Pipelines improve the best performance by 120%, with 21% less energy, or improve the best energy by 12% with 230% improvement in performance. In comparison with LI-BDNs, Fluid Pipelines improved the best performance by 33%, using 83% less energy, or improved the best energy by 38% with 118% better performance.

5.4 Elastic FPU

The maximum throughput for each of the models is summarized in Table 6.2. Fluid Pipelines deliver constant throughput regardless of the number of pipelines. The throughput of SELF decreases when there is additional pipeline stages in the sequential loops. In the case of LI-BDNs, the extra buffering helps maintaining the throughput even after the insertion of a few stages in the loops, but after a certain number of insertions, there is back pressure due to the dependencies.

The effective frequency, calculated for the average throughput, is reported in Fig. 6.18. It does not necessarily increase with the number of pipeline stages. This is due to the fact that despite the frequency gain with the new pipeline stage, the reduced throughput reverts the gains and reduces the overall performance. Since in the average case the loop path is used, there is a reduction in the gap between Fluid Pipelines and the other models. The same fact also causes reduction in the throughput of both SELF and LI-BDN. Despite the reduction in the gap, Fluid Pipelines are still able to deliver a considerably improved performance compared to SELF (120%), and slightly improved performance compared to LI-BDN (40%), but using less resources.

Table 6.2 The maximum expected throughput with respect to the number of pipeline stages is shown for the FPU design when using Fluid Pipelines, SELF and LI-BDNs. The original design contains six pipeline stages. Fluid Pipelines deliver constant maximum throughput, regardless of the number of pipeline stages

Pipeline stages	Fluid Pipelines	SELF	LI-BDN
6	1.00	1.00	1.00
7	1.00	1.00	1.00
8	1.00	1.00	1.00
9	1.00	0.67	1.00
10	1.00	0.50	1.00
11	1.00	0.40	0.83
12	1.00	0.37	0.74
13	1.00	0.33	0.67

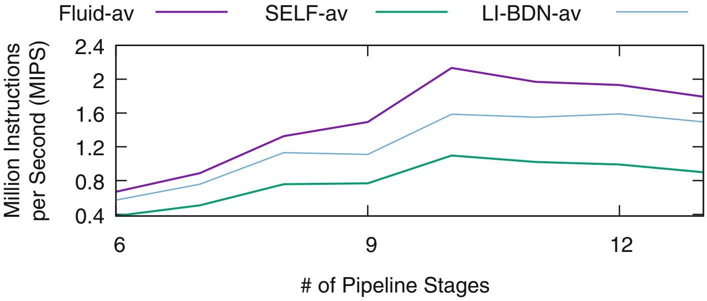


Fig. 6.18 The average throughput with respect to the number of pipeline stages is shown for the FPU for Fluid Pipelines, SELF, and LI-BDN. The average was calculated over a random input set. In Fluid Pipelines, circuits can be ReCycled with higher throughput than possible with Elastic Systems, and thus for better system performance

ED is reported in Fig. 6.19. The energy overhead caused by the extra storage in LI-BDNs reverses the advantages when compared to SELF. When comparing Fluid Pipelines with SELF, Fluid Pipelines improve the best ED point by improving performance by 176%, with 5% better energy. Alternatively, Fluid Pipelines deliver 120% better top performance (with 21% less energy). When comparing Fluid Pipelines with LI-BDNs, Fluid Pipelines improve the best ED point by improving both performance (by 163%) and energy (by 25%).

5.5 *Elastic OoO Core*

LI-BDNs were not considered, since their main improvement over SELF is the addition of FIFOs between modules. This is an important overhead for both area and power. In addition, we note from the previous experiment that for deep pipelining, LI-BDN behavior approaches that of SELF.

As in the FPU case, the effective frequency fluctuates (Fig. 6.20) when the frequency improvement is not enough to compensate for the throughput decrease.

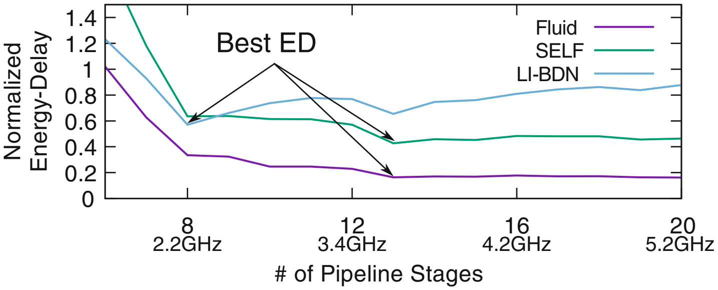


Fig. 6.19 Energy-delay product by frequency for the FPU design. The plot was made varying the number of pipeline stages and calculating the ED product and expected frequency for each pipeline configuration. Fluid Pipelines is shown to improve the best ED point of the FPU, pushing the depth of the pipeline

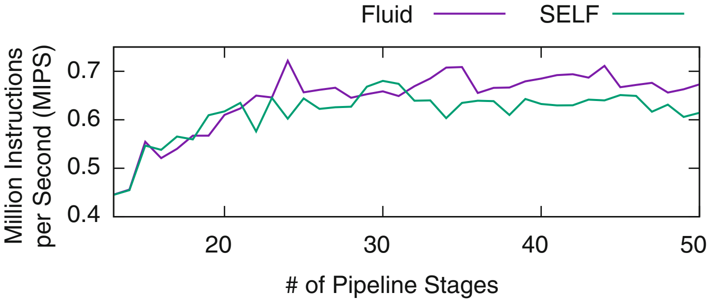


Fig. 6.20 The *plot* shows effective frequency, in million instruction per seconds (MIPS) for the OoO core. Effective frequency considers both throughput and frequency for each pipeline configuration. Nevertheless, effective frequency alone is not a fair metric since it does not consider the extra registers added by SELF

Note that for some points, SELF yields better overall performance than Fluid Pipelines. This is due to the insertion of extra pipeline stages into all the paths that are parallel to the critical path, which in some cases ends up hitting the second most critical path, and yields a better frequency increase, with a cost in power and area (area is not reported).

When adding extra pipeline stages, there is a initial phase where the insertion causes a considerable increase in frequency, with relatively small reduction on IPC (throughput) and increase in energy. This leads to an overall improvement in the ED (Fig. 6.21). As the pipeline depth increases, the addition of extra stages has a smaller impact on frequency, but with higher reduction on IPC which results in an overall degradation of ED. In other terms, a relatively high number of stages (i.e., power overhead) is needed to improve the overall performance, and thus ED gets worse. In SELF, when one stage is added to a path, the optimal solution for throughput is to also add a stage in all parallel paths with extra power overhead. Also in SELF, adding stages has a negative effect on throughput. Combining these two effects results in a faster degradation of ED. Fluid Pipelines shift the optimal

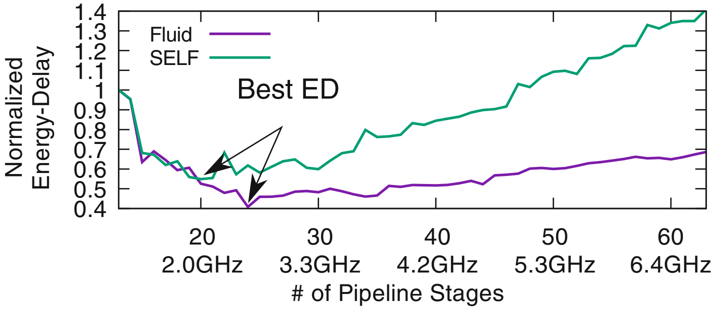


Fig. 6.21 The energy-delay product is shown for each pipeline configuration for the OoO core. The frequency shown in the x-axis was estimated based on the frequency for the original design and the number of added stages. The figure shows that Fluid Pipelines shift the optimal ED point of the pipeline depth and improve performance with a smaller power overhead

number of pipeline stages, make a deeper pipeline configuration, while improving energy by 13% and performance by 17%.

6 Conclusion

In this chapter, we present Fluid Pipelines, a new abstraction for Elastic Systems. By using Fluid Pipelines, the designer has the opportunity to extract OoO execution from the circuit whenever possible, and boost the design performance. Fluid Pipelines push the design's Pareto frontier, by improving performance and energy. In our experiments, Fluid Pipelines improve the optimal ED configuration of an OoO core by improving energy 13% and performance by 17%, over SELF. For a pure high performance configuration, Fluid Pipelines deliver 6% better top performance while using 28% less energy. In addition, Fluid Pipelines brings the advantages already existing in prior Elastic System approaches, like the possibility of changing the number of pipeline stages, without breaking the design functionality, and thus improves the ability of a designer to meet the design targets.

Fluid Pipelines can be used as a design strategy to generate multiple end-products. For instance, the same RTL can be used to generate a deep-pipelined high performance design and a design with few pipeline stages for low power. It is common for companies to keep multiple teams to create designs for each of those points. This practice leads to replication of work and code, that could be easily avoided with a Fluid Pipelines-oriented strategy.

We also present a modeling framework using Coloured Petri Nets, which allows designers to evaluate the system runtime behavior, and perform early design space exploration. This framework is later used to evaluate Fluid Pipelines against other Elastic System approaches, showing an improvement in the overall throughput of the systems.

Fluid Pipelines open many research opportunities in EDA and architecture alike. From a circuit designer perspective, Fluid Pipelines enable a more logic-oriented design methodology, less worried with physical design constraints. For architects, Fluid Pipelines provide a framework for flow control, opposed to the current token-credit approaches commonly used in CPU cores. Fluid Pipelines also allow for faster exploration of the design space and energy-delay trade-offs. But it is in EDA that Fluid Pipelines open the most interesting opportunities. A number of automated transformations is possible in Fluid Pipelines. We have discussed RePipelining (ReCycling + Retiming), but Fluid Pipelines transformations are not limited to it. It is also possible to apply resource utilization techniques of port sizing optimization and pipeline stage replication. For example, Fluid Pipelines allow to increase or decrease the number of ports required by an SRAM without changing overall system correctness. This is possible because when not enough ports are available at run-time, it is legal to stall the inputs and wait until a free port becomes available. As long as the stall operation is not frequent, the performance is not affected. Such transformation leverages the handshake signals of Fluid Pipelines under the hood to generate the proper control signals.

Acknowledgements This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

References

1. D. Baudisch, K. Schneider, Evaluation of speculation in out-of-order execution of synchronous dataflow networks. Int. J. Parallel Program. **43**(1), 86–129 (2015). doi:10.1007/s10766-013-0277-2
2. D. Bufistov, J. Cortadella, M. Galceran-Oms, J. Julvez, M. Kishinevsky, Retiming and recycling for elastic systems with early evaluation, in *46th Design Automation Conference* (2009), pp. 288–291
3. B. Cao, K. Ross, M. Kim, S. Edwards, Implementing latency-insensitive dataflow blocks, in *Proceedings of the 13th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'15* (2015)
4. L.P. Carloni, A.L. Sangiovanni-Vincentelli, Performance analysis and optimization of latency insensitive systems, in *Proceedings of the 37th Design Automation Conference* (ACM, New York, NY, 2000), pp. 361–367. doi:http://doi.acm.org/10.1145/337292.337441
5. L.P. Carloni, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, A methodology for correct-by-construction latency-insensitive design, in *International Conference on Computer-Aided Design* (1999), pp. 309–315
6. L.F. Chao, A. LaPaugh, E.M. Sha, Rotation scheduling: a loop pipelining algorithm. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **16**(3), 229–239 (1997). doi:10.1109/43.594829
7. N. Choudhary, B. Dwiel, E. Rotenberg, A physical design study of FabScalar-generated superscalar cores, in *2012 IEEE/IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)* (2012), pp. 165–170. doi:10.1109/VLSI-SoC.2012.6379024

8. J. Cortadella, M. Kishinevsky, B. Grundmann, SELF: specification and design of synchronous elastic circuits, in *Proceedings of the ACM/IEEE International Workshop on Timing Issues, TAU 06* (2006)
9. J. Cortadella, M. Galceran-Oms, M. Kishinevsky, Elastic systems, in *Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE '10* (2010), pp. 149–158
10. J. Cortadella, M. Galceran-Oms, M. Kishinevsky, S.S. Sapatnekar, Rtl synthesis: from logic synthesis to automatic pipelining. Proc. IEEE **103**(11), 2061–2075 (2015). doi:10.1109/JPROC.2015.2456189
11. G. Dimitrakopoulos, I. Seitanidis, A. Psarras, K. Tsiouris, P.M. Mattheakis, J. Cortadella, Hardware primitives for the synthesis of multithreaded elastic systems, in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)* (2014), pp. 1–4. doi:10.7873/DATE.2014.314
12. I. Ganusov, H. Fraisse, A. Ng, R.T. Possignolo, S. Das, Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs, in *Proceedings of the 26th Conference on Field Programmable Logic and Applications (FPL)* (2016)
13. M. Hrishikesh, D. Burger, N.P. Jouppi, K.I. Farkas, P. Shivakumar, The optimal logic depth per pipeline stage is 6 to 8 for inverter delays, in *Proceedings of the 29th International Symposium on Computer Architecture* (2002)
14. Y. Huang, P. Ienne, O. Temam, Y. Chen, C. Wu, Elastic CGRAs, in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (ACM, New York, NY, 2013), pp. 171–180. doi:10.1145/2435264.2435296
15. K. Jensen, L.M. Kristensen, *Coloured Petri Nets Modelling and Validation of Concurrent Systems* (Springer, Berlin, Heidelberg, 2009)
16. J. Julvez, J. Cortadella, M. Kishinevsky, Performance analysis of concurrent systems with early evaluation, in *International Conference on Computer-Aided Design*, pp. 448–455 (2006). doi:10.1109/ICCAD.2006.320155
17. K.E. Ardestani, J. Renau, ESESC: a fast multicore simulator using time-based sampling, in *International Symposium on High Performance Computer Architecture, HPCA'19* (2013)
18. C.E. Leiserson, J.B. Saxe, Retiming synchronous circuitry. Algorithmica **6**, 5–35 (1991)
19. S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, N. Jouppi, McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures, in *42nd IEEE/ACM Int'l Symp. on Microarchitecture* (IEEE, New York, 2009), pp. 469–480
20. M. Oskin, F. Chong, M. Farrens, HLS: combining statistical and symbolic simulation to guide microprocessor designs, in *International Symposium on Computer Architecture*, Vancouver (2000), pp. 71–82
21. R.T. Possignolo, E. Ebrahimi, H. Skinner, J. Renau, FluidPipelines: elastic circuitry meets out-of-order execution, in *Proceedings of the 34th International Conference on Computer Design (ICCD)* (2016)
22. R.T. Possignolo, E. Ebrahimi, H. Skinner, J. Renau, FluidPipelines: elastic circuitry without throughput penalty, in *Proceedings of the 2016 International Workshop on Logic Synthesis (IWLS)* (2016)
23. M. Vijayaraghavan, A. Arvind, Bounded dataflow networks and latency-insensitive circuits, in *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign* (IEEE, Piscataway, NJ, 2009), pp. 171–180

Chapter 7
Analysis of Incomplete Circuits Using Dependency Quantified Boolean Formulas

Ralf Wimmer, Karina Wimmer, Christoph Scholl, and Bernd Becker

1 Introduction

Solver-based techniques have proven to be successful in many areas in computer-aided design, ranging from formal verification of digital circuits [1, 3, 9, 29] over automatic test pattern generation [11, 13] to circuit synthesis [4, 5]. While research on solving quantifier-free Boolean formulas (the famous SAT-problem [10]) has reached a certain level of maturity, designing and improving algorithms for quantified Boolean formulas (QBFs) is one focus of active research. However, there are applications like the verification of partial circuits [18, 19, 29], the synthesis of safe controllers [4], and the analysis of games with incomplete information [26] for which QBF is not expressive enough to provide a compact and natural formulation. The reason is that QBF requires linearly ordered dependencies of the existential variables on the universal ones: Each existential variable implicitly depends on all universal variables in whose scope it is. Relaxing this condition yields so-called *dependency quantified Boolean formulas (DQBFs)*. DQBFs are strictly more expressive than QBFs in the sense that an equivalent QBF formulation can be exponentially larger than a DQBF formulation. This comes at the price of a higher complexity of the decision problem: DQBF is NEXPTIME-complete [26], compared to QBF, which is "only" PSPACE-complete. Encouraged by the success of SAT and QBF solvers and driven by the mentioned applications, research on solving DQBFs has started during the last few years [16, 17, 20, 33], yielding first prototypic solvers like IDQ [17] and HQS [20].

R. Wimmer (✉) • K. Wimmer • C. Scholl • B. Becker
Institute of Computer Science, Albert-Ludwigs-Universität Freiburg, Freiburg im Breisgau, Germany
e-mail: wimmer@informatik.uni-freiburg.de; wimmerka@informatik.uni-freiburg.de; scholl@informatik.uni-freiburg.de; becker@informatik.uni-freiburg.de

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_7

In this paper, we focus on the application of DQBF for analyzing *incomplete combinational and sequential circuits*. Such incomplete circuits appear in early design stages, when only a subset of the system's modules has already been implemented and verification is applied in order to find errors in the available parts as early as possible. Incomplete circuits also result if the complexity of the verification task is too high and therefore some parts, which are supposed not to influence the validity of some properties, e. g., multiplier or memory modules, have been removed to make verification feasible. Analyzing incomplete circuits is also useful if a designer wants to localize errors (then one can remove parts of the design and if for all possible implementations of the removed parts the error does not disappear, the remaining parts must be erroneous). Therefore this problem has received considerable attention in the research community during the last 15 years, see, e. g., [12, 14, 21, 25, 29–31]. All solver-based approaches are restricted in the sense that they can either only handle a single black box or do not take the interfaces of the black boxes into account, allowing the black boxes to read signals which are not available to them in the actual design.

We show how the realizability problem for incomplete combinational and sequential circuits with an arbitrary number of combinational or bounded-memory black boxes can be expressed as a DQBF. Here we show for the first time a DQBF-based solution for sequential circuits with several bounded-memory black boxes where the exact interface of the black boxes, i.e., the signals entering and leaving the black boxes, can be taken into account. We also show that solving a DQBF has the same complexity as deciding realizability. We do not only sketch how DQBFs are solved in our DQBF solver HQS [20, 33], but also how so-called Skolem functions can be obtained from the solution process, provided that the formula is satisfied [34]. These Skolem functions can directly serve as an implementation of the black boxes.

This paper builds on different sources: [18, 19] applies DQBF-based methods to incomplete combinational circuits with combinational black boxes. SAT- and QBF-based techniques for controller synthesis are considered in [4]; there a footnote gives hints how a DQBF formulation can be used for that purpose. Due to the lack of efficient DQBF solvers at that time, this idea was not investigated further. However the method described there considers only a single black box which can read all primary inputs and the complete state information. The basic techniques implemented in our DQBF solver HQS have been described in [20], and [33] defines preprocessing techniques for DQBF, which speed up the solution process considerably.

1.1 Structure of the Paper

In the next section, we introduce dependency quantified Boolean formulas (DQBFs). In Sect. 3, we describe how realizability of incomplete combinational and sequential circuits can be formulated as a DQBF. Section 4 presents a method to solve DQBFs and to obtain Skolem functions for satisfied DQBFs. In Sect. 5 we give preliminary experimental results, and we conclude the paper in Sect. 6, pointing out challenges which need to be solved.

2 Foundations

Let φ and κ be quantifier-free Boolean formulas over the set V of Boolean variables and $v \in V$. We denote by $\varphi[\kappa/v]$ the Boolean formula which results from φ by replacing all occurrences of v (simultaneously) by κ. For a set $V' \subseteq V$, we denote by $\mathcal{A}(V')$ the set of Boolean assignments for V', i.e., $\mathcal{A}(V') = \{\nu \mid \nu : V' \to \{0, 1\}\}$. For each quantifier-free formula φ over V, a variable assignment ν to the variables in V induces a truth value 0 or 1 of φ, which we call $\nu(\varphi)$.

Definition 1 (Syntax of DQBF) Let $V = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ be a set of Boolean variables. A *dependency quantified Boolean formula* (DQBF) ψ over V has the form $\psi := \forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_m(D_{y_m}) : \varphi$, where $D_{y_i} \subseteq \{x_1, \ldots, x_n\}$ for $i = 1, \ldots, m$ is the *dependency set* of y_i, and φ is a quantifier-free Boolean formula over V, called the *matrix* of ψ.

$V_\psi^\forall = \{x_1, \ldots, x_n\}$ denotes the set of universal and $V_\psi^\exists = \{y_1, \ldots, y_m\}$ the set of existential variables. We often write $\psi = Q : \varphi$ with the quantifier prefix Q and the matrix φ. $Q \setminus \{v\}$ denotes the prefix that results from removing a variable $v \in V$ from Q together with its quantifier. If v is existential, then its dependency set is removed as well; if v is universal, then all occurrences of v in the dependency sets of existential variables are removed. Similarly we use $Q \cup \{\exists y(D_y)\}$ to add existential variables to the prefix. We sometimes assume that a DQBF $\psi = Q : \varphi$ as in Definition 1 with φ in conjunctive normal form (CNF) is given. A formula is in CNF if it is a conjunction of (non-tautological) *clauses*; a clause is a disjunction of *literals*, and a literal is either a variable v or its negation $\neg v$. As usual, we identify a formula in CNF with its set of clauses and a clause with its set of literals. For a formula φ (resp. clause C, literal ℓ), $\mathrm{var}(\varphi)$ (resp. $\mathrm{var}(C)$, $\mathrm{var}(\ell)$) means the set of variables occurring in φ (resp. C, ℓ), $\mathrm{lit}(\varphi)$ ($\mathrm{lit}(C)$) means the set of literals occurring in φ (C).

A quantified Boolean formula (QBF) (in prenex normal form) is a DQBF such that $D_y \subseteq D_{y'}$ or $D_{y'} \subseteq D_y$ holds for any two existential variables $y, y' \in V_\psi^\exists$. Then the variables in V can be ordered resulting in a linear quantifier prefix, such that for each $y \in V_\psi^\exists$, D_y equals the set of universal variables which are to the left of y.

The semantics of a DQBF is usually defined by so-called Skolem functions.

Definition 2 (Semantics of DQBF) Let ψ be a DQBF as above. It is *satisfiable*, iff there are functions $s_y : \mathcal{A}(D_y) \to \mathbb{B}$ for $y \in V_\psi^\exists$ such that replacing each $y \in V_\psi^\exists$ by (a Boolean expression for) s_y turns φ into a tautology. The functions $(s_y)_{y \in V_\psi^\exists}$ are called *Skolem functions* for ψ.

Example 1 Consider the following DQBF:

$$\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : (x_1 \vee \neg y_1) \wedge (x_2 \vee y_1 \vee y_2)$$

Here the variable y_1 depends only on x_1, but not on x_2; y_2 depends only on x_2, but not on x_1. It is satisfied by using the Skolem functions $s_{y_1}(x_1) = x_1$ and $s_{y_2}(x_2) = \neg x_2$. Replacing y_1 and y_2 by their Skolem functions yields $(x_1 \vee \neg x_1) \wedge (x_2 \vee x_1 \vee \neg x_2)$, which is obviously a tautology.

3 Analysis of Incomplete Circuits

In this section, we show how DQBFs can be used to analyze incomplete combinational and sequential circuits. In both cases we ask for realizability: Are there implementations of the missing parts ("black boxes") such that the complete circuit satisfies its specification.

We assume that the missing parts are either combinational or contain only a bounded amount of memory. In the latter case, we can put the flipflops of the black boxes into the available circuit part such that the incoming and outgoing signals of these flipflops are written and read only by the black boxes as sketched in Fig. 7.1.

Then the black boxes themselves are purely combinational. Note that the case of several black boxes with an *unbounded* amount of memory is undecidable [28].

We use the notation for incomplete sequential circuits as sketched in Fig. 7.2. The primary inputs are denoted by $\mathbf{x}$, the current state by $\mathbf{s}$, and the next state by $\mathbf{s}'$. The missing parts are $\mathrm{BB}_1, \ldots, \mathrm{BB}_n$, whose interfaces, i.e., the signals entering and leaving the black boxes, are known. The input signals of black box BB_i are denoted by $\mathbf{I}_i$, its output signals by $\mathbf{y}_i$. The input cone of black box BB_i ensures the constraint $\mathbf{I}_i \equiv \mathbf{F}_i(\mathbf{s}, \mathbf{x}, \mathbf{y}_1, \ldots, \mathbf{y}_{i-1})$, the next state is described by trans :=

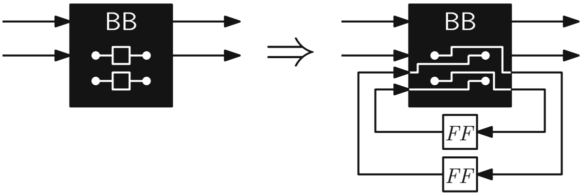


Fig. 7.1 Sequential circuits with extracted memory

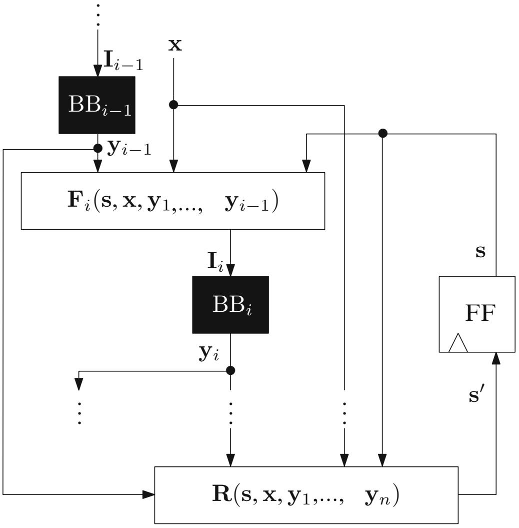

$\mathbf{I}_{i-1}$
x
BB_{i-1}
$\mathbf{y}_{i-1}$
$\mathbf{F}_i(\mathbf{s}, \mathbf{x}, \mathbf{y}_1,\ldots, \mathbf{y}_{i-1})$
$\mathbf{I}_i$
BB_i
$\mathbf{y}_i$
s
FF
s′
$\mathbf{R}(\mathbf{s}, \mathbf{x}, \mathbf{y}_1,\ldots, \mathbf{y}_n)$

Fig. 7.2 Notation for incomplete sequential circuits

$\big(\mathbf{s}' \equiv \mathbf{R}(\mathbf{s}, \mathbf{x}, \mathbf{y}_1, \ldots, \mathbf{y}_n)\big)$. We assume w. l. o. g. that no black box output is directly connected to an input of another black box or a flipflop, i.e., $\mathbf{y}_i \cap \mathbf{I}_j = \emptyset$ for all i, j and $\mathbf{s}' \cap \mathbf{y}_j = \emptyset$ for all j. Otherwise a buffer is inserted between the two black boxes without changing the functionality of the circuit. Additionally we assume that there are no cyclic dependencies between the combinational black boxes, i.e., that BB_i only depends on the outputs of $\mathrm{BB}_1, \ldots, \mathrm{BB}_{i-1}$. Otherwise, implementing the black boxes with combinational circuits can lead to cycles in the combinational part of the circuit which do not run through memory elements. This can cause undefined behavior of the circuit.

We consider invariant properties $\mathrm{inv}(\mathbf{s}, \mathbf{x}, \mathbf{y}_1, \ldots, \mathbf{y}_n)$, defined over the primary inputs $\mathbf{x}$, the current state $\mathbf{s}$ and the black box outputs $\mathbf{y}_1, \ldots, \mathbf{y}_n$, which are required to hold at any time.

3.1 Combinational Circuits

The same notation as introduced above is also used for combinational circuits. Here, the state and next state signals $\mathbf{s}$ and $\mathbf{s}'$ as well as the memory elements are omitted.

Definition 3 The *partial equivalence checking problem (PEC)* is defined as follows: Given an incomplete circuit C_{impl} and a (complete) specification C_{spec}, are there implementations of the black boxes in C_{impl} such that C_{impl} and C_{spec} become equivalent?

In the following, we assume that (incomplete) implementation C_{impl} and specification C_{spec} are combined into a single circuit using a miter construction: Corresponding primary inputs are connected, corresponding outputs are connected via XOR gates. The outputs of the XOR gates are combined via OR gates into a single output signal. This output signal is constantly one iff, for some implementation of the black boxes, the two circuits are equivalent. This can be considered as a kind of invariant property, valid at the primary output of the combined circuit.

We now show how a DQBF formulation can be used to decide PEC.

Consider a PEC problem with black boxes $\mathrm{BB}_1, \ldots, \mathrm{BB}_n$. We first construct the quantifier prefix of the DQBF. The primary inputs $\mathbf{x}$ and the black box inputs $\mathbf{I}_1, \ldots, \mathbf{I}_n$ are universally quantified, all other variables are existentially quantified. The dependency set of black box outputs $\mathbf{y}_i$ contains exactly the inputs $\mathbf{I}_i$ of BB_i. Hence the quantifier prefix is

$$\forall \mathbf{x} \forall \mathbf{I}_1 \ldots \forall \mathbf{I}_n \exists \mathbf{y}_1(\mathbf{I}_1) \ldots \exists \mathbf{y}_m(\mathbf{I}_m)\,.$$

If the black boxes are not directly connected to the primary inputs but to internal signals, we have to take into account that not all possible combinations of values may arrive at the inputs of the black boxes. Since we use a universal quantification for the black box inputs we have to ensure that our formula is satisfied if the value of the black box inputs $\mathbf{I}_i$ deviates from the values obtained as a function $F_i(\mathbf{x}, \mathbf{I}_1, \ldots, \mathbf{I}_{i-1})$.

$$\varphi(\mathbf{x}, \mathbf{I}_1, \ldots, \mathbf{I}_n, \mathbf{y}_1, \ldots, \mathbf{y}_n) := \left(\bigwedge_{i=1}^{n} \mathbf{I}_i \equiv \mathbf{F}_i \right) \Rightarrow \mathrm{inv}(\mathbf{x}, \mathbf{y}_1, \ldots, \mathbf{y}_n) \, .$$

This formula is not necessarily given in CNF. By applying Tseitin transformation [32], which is essentially introducing auxiliary variables for the internal signals of the circuit, one can obtain a CNF φ' that is equisatisfiable to φ and whose size is linear in the size of φ. Let **a** be the vector of these auxiliary variables, which are existentially quantified in the quantifier prefix. As their values are implied by the values of the variables in their input cone, we can use as their dependency sets either the universal variables in their input cone or the set of all universal variables (or any set in between). We prefer making the Tseitin variables depend on all universal variables, because this typically leads to DQBFs that are easier to solve.

The resulting DQBF is:

$$\psi := \forall \mathbf{x} \forall \mathbf{I}_1 \ldots \forall \mathbf{I}_n \exists \mathbf{y}_1(\mathbf{I}_1) \ldots \exists \mathbf{y}_n(\mathbf{I}_n) \exists \mathbf{a}(\mathbf{x}, \mathbf{I}_1, \ldots, \mathbf{I}_n) : \\ \varphi'(\mathbf{x}, \mathbf{I}_1, \ldots, \mathbf{I}_n, \mathbf{y}_1, \ldots, \mathbf{y}_n, \mathbf{a}) \, .$$

This formula ψ is satisfied iff we can replace all $\mathbf{y}_i(\mathbf{I}_i)$ with Skolem functions $s_i(\mathbf{I}_i)$ such that φ' becomes a tautology. The Skolem functions s_i exist if and only if there are implementations for the black boxes BB_i of the PEC, such that the PEC is satisfied. Therefore any PEC can be translated with linear effort into a DQBF and the PEC is satisfied iff the DQBF is satisfied.

It is easy to see that there is also the converse transformation [19]: Each DQBF can be turned into a PEC, having one black box for each existential variable such that the PEC is realizable iff the DQBF is satisfiable. This implies that PEC is NEXPTIME-complete.

3.2 *Sequential Circuits*

For incomplete sequential circuits with multiple combinational or bounded-memory black boxes, we investigate the following problem:

Definition 4 The *realizability problem for incomplete sequential circuits (RISC)* is defined as follows: Given an incomplete sequential circuit with multiple combinational (or bounded-memory) black boxes and an invariant property, are there implementations of the black boxes such that in the complete circuit the invariant holds at all times?

To decide RISC, one can apply a generalization of ideas described in [4] for the synthesis of controllers (which are in fact single black boxes with access to all state bits and all primary circuit inputs).

According to the notations introduced in the previous section, let $\mathbf{s}$ denote the variables encoding the current state of the circuit, $\mathbf{s}'$ the next state, and $\mathbf{x}$ the primary inputs. The formulas $F_1, \ldots, F_n$ describe the input cones of the black boxes, $\mathbf{I}_1, \ldots, \mathbf{I}_n$ their inputs, $\mathbf{y}_1, \ldots, \mathbf{y}_n$ their outputs, and R is the next state function of the circuit. Additionally we assume that init represents the circuit's initial state(s) and inv its states that satisfy the invariant.

Definition 5 A subset $W \subseteq S$ is a *winning set* if all states in W satisfy the invariant and, for all values of the primary inputs, the black boxes can ensure (by computing appropriate values) that the successor state is again in W.

A given RISC is realizable if there is a winning set that includes the initial state of the circuit. This can be formulated as a DQBF. To encode the winning sets, we introduce two existential variables w and w'; w depends on the current state and is supposed to be true for a state $\mathbf{s}$ if $\mathbf{s}$ is in the winning set. The variable w' depends on the next state variables $\mathbf{s}'$ and has the same Skolem function as w (but defined over $\mathbf{s}'$ instead of $\mathbf{s}$). We ensure that w and w' have the same semantics by the condition $\big(\mathbf{s} \equiv \mathbf{s}' \Rightarrow w \equiv w'\big)$.

Similar to the combinational case, we have to take into account that the black boxes are typically not directly connected to the primary inputs, but to internal signals. This is done by restricting the requirement that the successor state is again a winning state to the case when the black box inputs are assigned consistently with the values computed by their input cones.

Theorem 1 *Given a RISC as defined above, the following DQBF is satisfied if and only if the RISC is realizable:*

$$\begin{aligned}\forall \mathbf{s}\, \forall \mathbf{s}' \forall \mathbf{x}\, \forall \mathbf{I}_1 \ldots \forall \mathbf{I}_n\, \exists \mathbf{y}_1(\mathbf{I}_1) \ldots \exists \mathbf{y}_n(\mathbf{I}_n)\, \exists w(\mathbf{s})\, \exists w'(\mathbf{s}') :&\\ \big(\text{init} \Rightarrow w\big) \wedge \big(w \Rightarrow \text{inv}\big) \wedge \big(\mathbf{s} \equiv \mathbf{s}' \Rightarrow w \equiv w'\big)&\\ \wedge \Big(\Big(w \wedge \bigwedge_{i=1}^{n} \mathbf{I}_i \equiv \mathbf{F}_i \wedge \mathbf{s}' \equiv R\Big) \Rightarrow w'\Big).&\end{aligned}$$

Theorem 2 *RISC is NEXPTIME-complete.*

Proof The reduction to DQBF above shows that RISC is in NEXPTIME. To show the hardness, we give a reduction from DQBF to RISC. First we transform the DQBF into an incomplete combinational circuit as shown in [18] such that the output of the circuit is constantly 1 iff the DQBF is satisfied. We now turn this combinational circuit into a sequential circuit with two states by storing the output of the combinational circuit in a 1-bit flipflop s. The initial state is $s \equiv 1$, the invariant is given by $s \equiv 1$. The original DQBF is satisfied iff the unsafe state 0 can be made unreachable by appropriate black box implementations. □

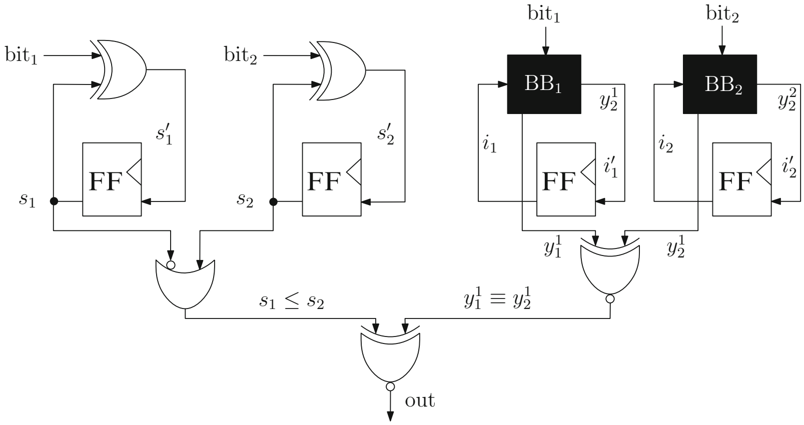


Fig. 7.3 Sequential circuit with *two black boxes*

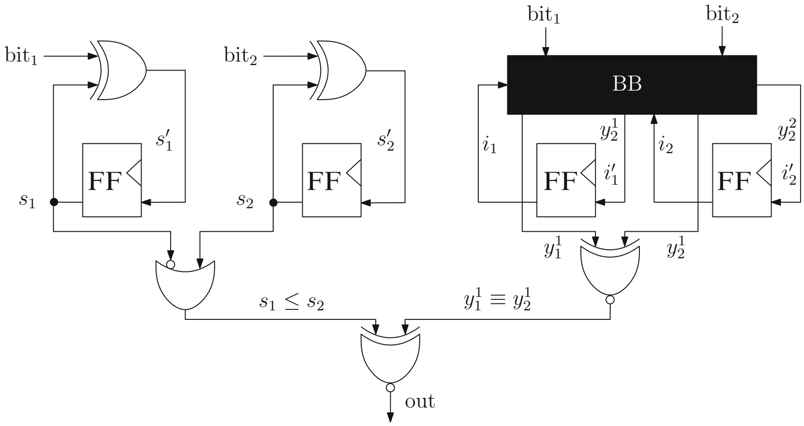


Fig. 7.4 The same sequential circuit as in Fig. 7.3, but with a *single black box*

Example 2 We illustrate the solution of RISC using two incomplete circuits in Figs. 7.3 and 7.4. The circuits are simple, but still illustrate the basic idea. We first start with the circuit in Fig. 7.3. The sequential circuit in Fig. 7.3 consists of two parts. The first part on the left can be seen as the specification for a simple sequential circuit: There are two bit streams applied to the inputs bit₁ and bit₂. The circuit computes the parities of the bit streams applied to bit₁ and bit₂ and outputs 1 iff the parity for bit stream bit₁ is smaller or equal to the parity for bit stream bit₂. The right-hand side shows a given architecture for an implementation with two black boxes, one reading bit stream bit₁ and the other reading bit stream bit₂. The outputs of the black boxes are connected by an equivalence gate. Then the output of the overall circuit is computed by an equivalence gate connecting the outputs of specification and incomplete implementation. We require the invariant property that the output of the overall circuit is 1 at all times, i.e., that the black boxes are implemented in a way such that the implementation part agrees with the specification part. For this simple example it is easy to see that a corresponding implementation does not exist, even for black boxes with unbounded memory. Here we use our method where the number of flip flops for each black box is restricted to one. Figure 7.3 already shows the transformed circuit where the memory is extracted from the black boxes.

Applying Theorem 1, we obtain the following formula parts:

- initial state:

$$\text{init} := (\neg s_1 \wedge \neg s_2 \wedge \neg i_1 \wedge \neg i_2)$$

- transition relation:

$$\begin{aligned}\text{trans} := (s_1' \equiv s_1 \oplus \text{bit}_1) \wedge (s_2' \equiv s_2 \oplus \text{bit}_1)\\ \wedge (i_1' \equiv y_2^1) \wedge (i_2' \equiv y_2^2)\end{aligned}$$

- invariant:

$$\text{inv} := \big(\neg s_1 \vee s_2\big) \equiv (y_1^1 \equiv y_2^1)\big)$$

Putting these parts together yields the following DQBF:

$$\begin{aligned}&\forall \text{bit}_1 \forall \text{bit}_2 \forall s_1 \forall s_1' \forall s_2 \forall s_2' \forall i_1 \forall i_1' \forall i_2 \forall i_2'\\ &\qquad \exists y_1^1(i_1, \text{bit}_1) \exists y_2^1(i_1, \text{bit}_1) \exists y_1^2(i_2, \text{bit}_2) \exists y_2^2(i_2, \text{bit}_2)\\ &\qquad \exists w(s_1, s_2, i_1, i_2) \exists w'(s_1', s_2', i_1', i_2') :\\ &\qquad (\text{init} \Rightarrow w) \wedge (w \Rightarrow \text{inv}) \wedge \big((w \wedge \text{trans}) \Rightarrow w'\big)\\ &\qquad \wedge \big(((s_1 \equiv s_1') \wedge (s_2 \equiv s_2') \wedge (i_1 \equiv i_1') \wedge (i_2 \equiv i_2')) \Rightarrow (w \equiv w')\big)\end{aligned}$$

By applying a DQBF solver, one can verify that this formula is unsatisfiable, meaning that the design in Fig. 7.3 is not realizable.

Now consider the circuit in Fig. 7.4. It differs from the design in Fig. 7.3 only in the black boxes: Both black boxes can read both input signals bit_1 and bit_2. Thus, the black boxes can equivalently be merged into one as shown in Fig. 7.4. It is easy to see that this implementation, which does not pay attention to the exact architecture by disregarding the interface of the black boxes, is now realizable. More precisely, it is realizable if we assume that the black box with bounded memory has at least 2 memory cells at its disposal. In Fig. 7.4 we depict the incomplete circuit with two memory cells extracted from the black box. Using our approach we can indeed prove realizability. The formula differs only in the dependency sets of the black box outputs: $D_{y_1^1} = D_{y_2^1} = D_{y_1^2} = D_{y_2^2} = \{\mathrm{bit}_1, \mathrm{bit}_2, i_1, i_2\}$. Now the formula is satisfiable. The following Skolem functions turn the matrix into a tautology:

Variable	y_1^1	y_2^1	y_1^2	y_2^2	w	w'
Skolem function	1	$\mathrm{bit}_1 \oplus i_1$	$\neg i_1 \vee i_2$	$\mathrm{bit}_2 \oplus i_2$	1	1

Using these Skolem functions, the two flipflops in the right half store the same values as the two flipflops in the left half, i.e., $s_1 = i_1$ and $s_2 = i_2$. The equivalence $y_1^1 \equiv y_2^1$ corresponds to $1 \equiv (\neg i_1 \vee i_2)$, which is the same as $i_1 \leq i_2$. Therefore the design is realizable.

For both incomplete circuits, our solver HQS [20] solved the DQBF in at most 0.1 s.

We can conclude that it is necessary to take the precise interfaces of the black boxes into account in order to obtain a valid answer whether the design is realizable.

4 Solving DQBFs

Elimination-based DQBF solvers like HQS [20, 33] apply a series of satisfiability-preserving transformation steps to the formula until a SAT or QBF problem results, which can be solved by an arbitrary SAT or QBF solver. As a pure yes/no answer is not satisfactory when solving analysis problems as presented in the previous section, we provide the main ideas how Skolem functions can be extracted from the solution process. More details can be found in [34, 35]. This extraction proceeds in the reverse order of transformation, starting with (constant) Skolem functions for the final SAT problem, which correspond to a satisfying assignment.

4.1 Transformation Steps

The central operation of elimination-based solvers is the elimination of existential and universal variables from the formula. QBF solvers can eliminate variables in the order given by the quantifier prefix (starting with the inner-most variable block). Because there is no linear order on the variables in a DQBF, this is typically no longer possible.

Lemma 1 *Let $\psi = Q : \phi$ be a DQBF and $y \in V_\psi^\exists$ an existential variable which depends on all universal variables. Then ψ is equisatisfiable to $\psi' := Q \setminus \{\exists y(D_y)\} : \phi[0/y] \vee \phi[1/y]$.*

If $s_z^{\psi'}$ for $z \in V_{\psi'}^{\exists}$ are Skolem functions for the formula ψ', obtained by eliminating $y \in V_{\psi}^{\exists}$, we set $s_y^{\psi} := \phi[1/y][s_z^{\psi'}/z]$ and $s_z^{\psi} := s_z^{\psi'}$ for $z \neq y$. This yields Skolem functions for ψ [34].

The elimination of universal variables in solvers like HQS [20] is done by *universal expansion* [2, 7, 8, 19]. This is applicable even if some existential variables depend on the expanded universal one.

Lemma 2 *For a DQBF* $\psi = \forall x_1 \dots \forall x_n \exists y_1(D_{y_1}) \dots \exists y_m(D_{y_m}) : \varphi$ *with* $E_{x_i} = \big\{y_j \in V_{\psi}^{\exists} \,\big|\, x_i \in D_{y_j})\big\}$, *the* universal expansion *w. r. t. variable* $x_i \in V_{\psi}^{\forall}$, *is defined by*

$$\big(Q \setminus \{x_i\}\big) \cup \big\{\exists y_j'(D_{y_j} \setminus \{x_i\}) \,\big|\, y_j \in E_{x_i}\big\} : \qquad \varphi[1/x_i] \wedge \varphi[0/x_i][y_j'/y_j \textit{ for all } y_j \in E_{x_i}]\,.$$

That means, when eliminating a universal variable $x \in V_{\psi}^{\forall}$, we have to copy all existential variables $y \in V_{\psi}^{\exists}$ that depend on x.

Assume that $s_z^{\psi'}$ for $z \in V_{\psi'}^{\exists}$ are Skolem functions for ψ'. Then $s_y^{\psi} := (x \wedge s_{y'}^{\psi'}) \vee (\neg x \wedge s_y^{\psi'})$ for $y \in V_{\psi}^{\exists}$ with $x \in D_y$ and $s_z^{\psi} := s_y^{\psi'}$ for $z \in V_{\psi}^{\exists}$ with $x \notin D_z$ are Skolem functions for ψ [34].

In principle, these two operations suffice to turn each DQBF into an (exponentially larger) SAT problem. In order to reduce computation time and memory consumption, pre- and inprocessing steps have turned out to be essential.

Standard operations are the detection of unit and pure literals. A literal ℓ is unit if (ℓ) is a clause in the formula. A literal ℓ is pure if $\neg\ell$ does not appear in the formula. In both cases $\mathrm{var}(\ell)$ can be replaced by a constant (which is also the Skolem function for that variable). Further preprocessing techniques like blocked clause elimination, the identification of equivalent variables, and structure extraction have been devised for DQBF [33, 36]. All of them are supported when Skolem functions are to be computed. We refer to [34] for more details.

4.2 Elimination Sets

Since the expansion of all universal variables leads to an exponentially larger SAT instance, this is typically not feasible. Instead, the solver HQS eliminates variables only until a QBF is obtained, which can be solved by an arbitrary QBF solver. The central problem is to determine a minimum set of universal variables whose elimination turns the DQBF into a QBF [20]. To solve this, we can use the following dependency graph:

Definition 6 Let ψ be a DQBF. Its *dependency graph* $G_\psi = (V_\psi^\exists, E_\psi)$ is a directed graph with the existential variables as its nodes and edges $E_\psi = \{(y,z) \in V_\psi^\exists \times V_\psi^\exists \mid D_y \not\subseteq D_z\}$.
It can be used to recognize if a DQBF is actually a QBF:

Lemma 3 *Let ψ be a DQBF. Its dependency graph G_ψ is acyclic iff ψ has an equivalent QBF prefix.*
That means we have to find a minimum set $U \subseteq V_\psi^\forall$ of universal variables whose elimination makes G_ψ acyclic. One can show that for making the graph acyclic by eliminating universal variables, it suffices to consider the cycles of length 2. The selection of variables can be done using a MAXSAT solver: for each universal variable x a variable m_x is created in the MAXSAT solver such that $m_x = 1$ means that x needs to be eliminated. Soft clauses are used to get an assignment with a minimum number of variables assigned to 1. Hard clauses enforce that for all $y, z \in V_\psi^\exists$, $y \neq z$, either all variables in $D_y \setminus D_z$ or in $D_z \setminus D_y$ are eliminated.

The variables in U are then eliminated, ordered according to the number of existential variables that depend upon them.

For more details, including formal correctness proofs, we refer the reader to [20].

4.3 Solver Overview

Figure 7.5 shows the structure of the general-purpose DQBF solver HQS. The input is a DQBF in CNF. After preprocessing, which is done on the CNF, gate detection is applied, essentially undoing Tseitin transformation and removing the existential variables introduced by the CNF transformation. The result is a representation of the

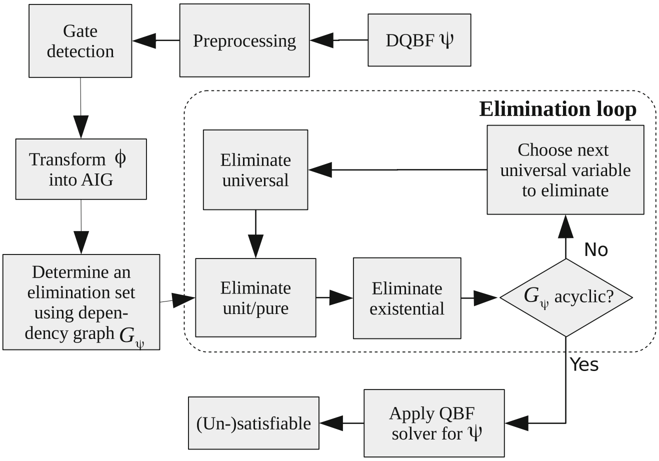


Fig. 7.5 Structure of the solver (adapted from [20])

formula as an And-Inverter graph (AIG), on which the further steps are performed. Before the actual elimination loop starts, we determine a minimum elimination set as described above.

Within the elimination loop, we check for unit and pure variables, which can be replaced by constants. This is done on the AIG using syntactic checks. Additionally, all existential variables are eliminated for which this is possible. Otherwise they would double for each eliminated universal variable. Then we check if the dependency graph has already become acyclic. If this is the case we generate the corresponding QBF prefix and solve the formula using the QBF solver AIGsolve [27], which operates directly on AIGs. Otherwise we select the next universal variable to eliminate and expand it.

5 Experimental Results

In the following, we present preliminary experimental results for incomplete combinational circuits. To solve the DQBFs, we use our elimination-based DQBF solver HQS [20], which was described briefly in the previous section.

We have extended HQS by the possibility to compute Skolem functions for satisfied DQBFs. The computation of Skolem functions works in two phases: During the solution process we collect the necessary data and store it on a stack. When the satisfiability of the formula has been determined, we free the other data structures of the solver and extract the Skolem functions from the collected data. We can apply don't-care minimization to the Skolem functions, based on Craig interpolants [24], and use the tool ABC [6] for further minimization of the Skolem functions' AIG representation.

All experiments were run on one Intel Xeon E5-2650v2 CPU core at 2.60 GHz clock frequency and 64 GB of main memory under Ubuntu Linux as operating system. We aborted all experiments which either took more than 1000 s CPU time or more than 8 GB ($= 2^{30}$ bytes) of main memory. As benchmarks we used 4318 DQBF instances from different sources. Most of them are DQBFs resulting from equivalence checking of incomplete combinational circuits [15, 17, 19]. The remaining ones are controller synthesis problems [4]. The sizes of the circuits range from a few hundred to a few thousand gates. The incomplete circuits contain one to five randomly selected black boxes. The controller synthesis problems are essentially sequential circuits with a single black box.

We first compare the efficiency of HQS with the only other available DQBF solver iDQ [17], which solves the formula by iteratively solving SAT instances generated from the DQBF. Both solvers were run after preprocessing the DQBFs. Since iDQ relies on a formula in CNF, while HQS does not, different preprocessing operations had to be applied: besides standard techniques like the detection of unit and pure literals and equivalent variables, preprocessing for HQS applies gate detection, which reverses Tseitin transformation in order to reconstruct the original circuit. This is not possible in case of iDQ. Instead, for iDQ, we apply blocked clause elimination and variable elimination by resolution, which are both not beneficial for HQS. We refer the reader to [33, 36] for more details on DQBF preprocessing.

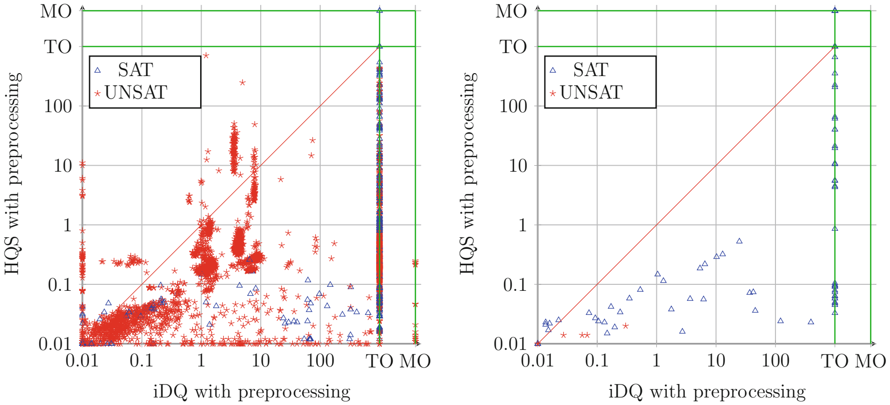


Fig. 7.6 Computation times (in seconds) of HQS and iDQ (both with preprocessing) on PEC instances [15, 17, 20] (*left*) and instances from controller synthesis [4] (*right*)

Figure 7.6 shows the results for incomplete combinational circuits (left, 3686 instances) and instances from controller synthesis (right, 89 instances) for those instances that could be solved by at least one solver; the remaining 543 instances could not be solved. The controller synthesis instances are incomplete sequential circuits with a single black box that can read all state bits and all primary inputs. We can observe in both cases, that HQS (with few exceptions) is more efficient and solves considerably more instances than iDQ. A closer look shows that the computation time and memory consumption is strongly influenced by the number of universal variables which have to be eliminated in order to obtain an equisatisfiable QBF. This is caused by the copies of the existential variables that are created when eliminating universal variables.

In spite of the improvements made during the last few years, the size of the instances that can effectively be solved is smaller by roughly one to two orders of magnitude than solvable QBF instances—strongly dependent on the number of variable copies which are created to obtain an equisatisfiable QBF.

The second set of experiments concerns the computation of Skolem functions for satisfiable DQBFs. We first measured the overhead of collecting the necessary data for computing Skolem functions during the solution process, i.e., until the truth value of the formula has been determined. The results are shown in Fig. 7.7. We can observe that the overhead is in most cases negligible—in a very few cases, the memory consumption is even reduced. The reason for this behavior is that within the AIG package different optimizations like rewriting are triggered when certain thresholds are exceeded. This can lead to smaller AIGs and thus save memory (and computation time for the subsequent operations).

For all 720 satisfiable instances we were able to solve without Skolem functions, we could also compute Skolem functions. For these instances we compare the sizes of the Skolem functions with and without optimizations using interpolation and ABC. Figure 7.8 displays the results. In many cases, we can reduce the sizes of the Skolem functions considerably, sometimes by up to two orders of magnitude.

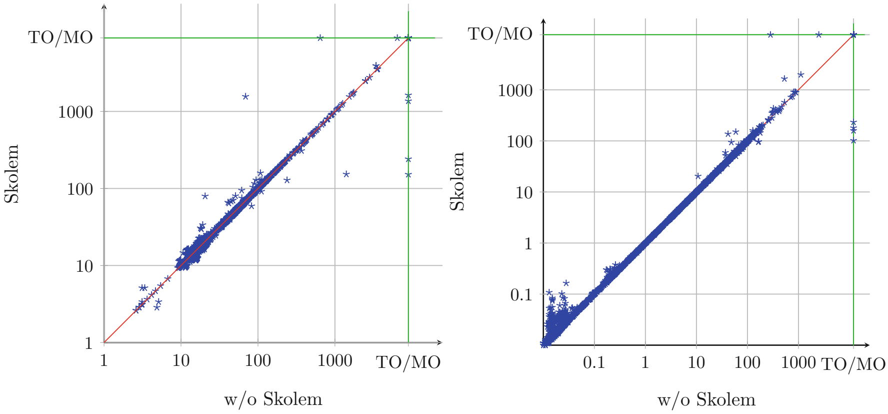
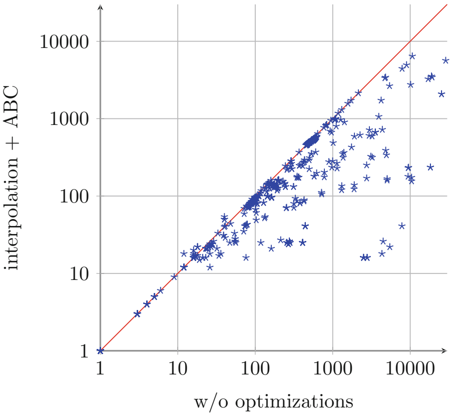


Fig. 7.7 Overhead of Skolem function computation on memory consumption (*left*) and running time (*right*)

Fig. 7.8 Size of the computed Skolem functions with and without optimizations



Because QBFs are a special case of DQBFs, we can use HQS to compute Skolem functions for satisfied QBFs as well. In Fig. 7.9, we compare the sizes of the Skolem functions generated by HQS with those generated by the state-of-the-art QBF solver DepQBF 5.0 [22, 23] for a set of satisfiable QBF instances from the QBF Gallery 2013[1] and from partial equivalence checking [29] (with a single black box). Since HQS (and in particular its preprocessor) is not optimized for solving QBF instances, we abstain from a detailed comparison of the running times of HQS and DepQBF. DepQBF is often (but not always) faster than HQS. In a few cases, the generation of Skolem functions with DepQBF failed, because the necessary resolution proof became too large (we aborted DepQBF when the size of the dumped resolution proof exceeded 20 GB).

[1] See http://www.kr.tuwien.ac.at/events/qbfgallery2013/.

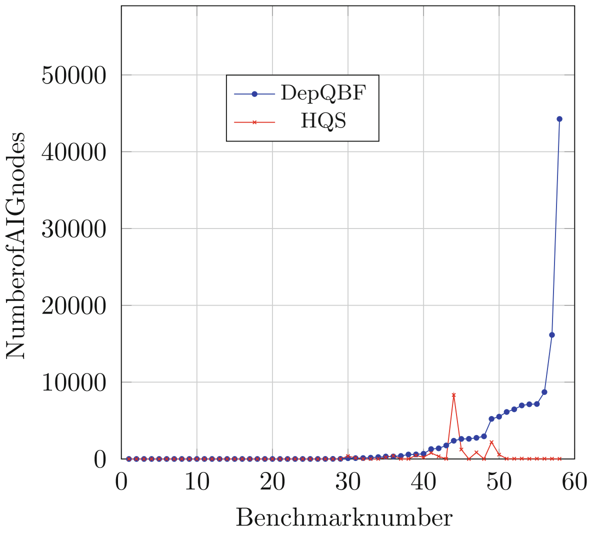


Fig. 7.9 Comparison of the sizes of Skolem functions from HQS and from DepQBF on QBF instances. The instances are ordered according to the size of DepQBF's Skolem functions

Figure 7.9 shows the sizes of the Skolem functions computed by DepQBF and by HQS (with interpolation and ABC). To enable a fair comparison, we also applied ABC with the same commands to the Skolem functions generated using DepQBF. We can observe that HQS' Skolem functions are in most cases smaller (often significantly) than those obtained from DepQBF.

In summary, the experiments show that HQS is able to solve the DQBFs resulting from small to medium-sized circuits effectively. We can not only obtain a pure yes/no answer, but also Skolem functions for the satisfiable instances without significant overhead. On satisfiable *QBF* instances, the size of the Skolem functions computed by HQS is similar, in many cases smaller in comparison to Skolem functions computed by DepQBF.

As a side remark, the example provided in Sect. 3 can easily be solved with a recent DQBF solver within fractions of a second. Benchmarks dealing with the synthesis of multiple black boxes in sequential circuits currently do not exist, but would be interesting to have.

6 Conclusion and Open Challenges

This paper has shown that DQBF formulations allow to express the realizability of invariant properties for incomplete combinational and sequential circuits with an arbitrary number of black boxes in a natural way. First prototypic solvers allow not only to solve the resulting DQBFs for small to medium-sized circuits, but also to extract Skolem functions, which can serve as implementations of the missing parts.

Still, many challenges remain: The scalability of the solvers has to be improved and might be tuned towards specific applications. More powerful preprocessing techniques are necessary as well as improvements in the solver core. We hope that with the availability of solvers more applications of these techniques become feasible (distributed controller synthesis) or are newly discovered thereby inspiring further improvements of the solvers—just as it was for propositional SAT solving and is for QBF solving.

References

1. P. Ashar, M.K. Ganai, A. Gupta, F. Ivancic, Z. Yang, Efficient SAT-based bounded model checking for software verification, in *International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, ed. by T. Margaria, B. Steffen, A. Philippou, M. Reitenspieß, Technical Report, Paphos, Cyprus, vol. TR-2004-6 (Department of Computer Science, University of Cyprus, 2004), pp. 157–164
2. V. Balabanov, H.-J. Katherine Chiang, J.-H.R. Jiang, Henkin quantifiers and Boolean formulae: a certification perspective of DQBF. Theor. Comput. Sci. **523**, 86–100 (2014)
3. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, Y. Zhu, Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
4. R. Bloem, R. Könighofer, M. Seidl, SAT-based synthesis methods for safety specs, in *Proceedings of VMCAI*, ed. by K.L. McMillan, X. Rival. Lecture Notes in Computer Science, San Diego, CA, vol. 8318 (Springer, Berlin, 2014), pp. 1–20
5. R. Bloem, U. Egly, P. Klampfl, R. Könighofer, F. Lonsing, M. Seidl, Satisfiability-based methods for reactive synthesis from safety specifications. CoRR, abs/1604.06204 (2016), http://arxiv.org/abs/1604.06204
6. R.K. Brayton, A. Mishchenko, ABC: an academic industrial-strength verification tool, in *Proceedings of CAV*, ed. by T. Touili, B. Cook, P. Jackson. Lecture Notes in Computer Science, Edinburgh, vol. 6174 (Springer, Berlin, 2010), pp. 24–40
7. U. Bubeck, Model-based transformations for quantified Boolean formulas. PhD thesis, University of Paderborn (2010)
8. U. Bubeck, H. Kleine Büning, Dependency quantified Horn formulas: models and complexity, in *Proceedings of SAT*, ed. by A. Biere, C.P. Gomes. Lecture Notes in Computer Science, Seattle, WA, vol. 4121 (Springer, Berlin, 2006), pp. 198–211
9. E.M. Clarke, A. Biere, R. Raimi, Y. Zhu, Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001)
10. S.A. Cook, The complexity of theorem-proving procedures, in *Proceedings of STOC* (ACM, New York, 1971), pp. 151–158
11. A. Czutro, I. Polian, M.D.T. Lewis, P. Engelke, S.M. Reddy, B. Becker, Thread-parallel integrated test pattern generator utilizing satisfiability analysis. Int. J. Parallel Prog. **38**(3–4),185–202 (2010)
12. W. Damm, B. Finkbeiner, Automatic compositional synthesis of distributed systems, in *Proceedings of FM*, ed. by C.B. Jones, P. Pihlajasaari, J. Sun. Lecture Notes in Computer Science, Singapore, vol. 8442 (Springer, Berlin, 2014), pp. 179–193
13. S. Eggersglüß, R. Drechsler, A highly fault-efficient SAT-based ATPG flow. IEEE Des. Test Comput. **29**(4), 63–70 (2012)
14. B. Finkbeiner, S. Schewe, Bounded synthesis. Int. J. Softw. Tools Technol. Transfer **15**(5–6), 519–539 (2013)
15. B. Finkbeiner, L. Tentrup, Fast DQBF refutation, in *Proceedings of SAT*, ed. by C. Sinz, U. Egly. Lecture Notes in Computer Science, Vienna, vol. 8561 (Springer, Berlin, 2014), pp. 243–251

16. A. Fröhlich, G. Kovásznai, A. Biere, A DPLL algorithm for solving DQBF, in *International Workshop on Pragmatics of SAT (POS)*, Trento (2012)
17. A. Fröhlich, G. Kovásznai, A. Biere, H. Veith, iDQ: instantiation-based DQBF solving, in *International Workshop on Pragmatics of SAT (POS)*, ed. by D. Le Berre. EPiC Series, Vienna, vol. 27 (EasyChair, 2014), pp. 103–116
18. K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, B. Becker, Equivalence checking for partial implementations revisited, in *Proceedings of MBMV*, ed. by C. Haubelt, D. Timmermann, Rostock (Universität Rostock, ITMZ, 2013), pp. 61–70
19. K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, B. Becker, Equivalence checking of partial designs using dependency quantified Boolean formulae, in *Proceedings of ICCD*, Asheville, NC (IEEE CS, 2013), pp. 396–403
20. K. Gitina, R. Wimmer, S. Reimer, M. Sauer, C. Scholl, B. Becker, Solving DQBF through quantifier elimination, in *Proceedings of DATE*, Grenoble (IEEE, New York, 2015)
21. M. Herbstritt, B. Becker, C. Scholl, Advanced SAT-techniques for bounded model checking of blackbox designs, in *Proceedings of MTV* (IEEE, New York, 2006), pp. 37–44
22. F. Lonsing, A. Biere, DepQBF: a dependency-aware QBF solver. J. Satisf. Boolean Model. Comput. **7**(2–3), 71–76 (2010)
23. F. Lonsing, F. Bacchus, A. Biere, U. Egly, M. Seidl, Enhancing search-based QBF solving by dynamic blocked clause elimination, in *Proceedings of LPAR*, ed. by M. Davis, A. Fehnker, A. McIver, A. Voronkov. Lecture Notes in Computer Science, Suva, vol. 9450 (Springer, Berlin, 2015), pp. 418–433
24. K.L. McMillan, Applications of Craig interpolants in model checking, in *Proceedings of TACAS*, ed. by N. Halbwachs, L.D. Zuck. Lecture Notes in Computer Science, Edinburgh, vol. 3440 (Springer, Berlin, 2005), pp. 1–12
25. T. Nopper, C. Scholl, Symbolic model checking for incomplete designs with flexible modeling of unknowns. IEEE Trans. Comput. **62**(6), 1234–1254 (2013)
26. G. Peterson, J. Reif, S. Azhar, Lower bounds for multiplayer non-cooperative games of incomplete information. Comput. Math. Appl. **41**(7–8), 957–992 (2001)
27. F. Pigorsch, C. Scholl, Exploiting structure in an AIG based QBF solver, in *Proceedings of DATE* (IEEE, New York, 2009), pp. 1596–1601
28. A. Pnueli, R. Rosner, Distributed reactive systems are hard to synthesize, in *Annual Symposium on Foundations of Computer Science*, St. Louis, MO (IEEE Computer Society, Washington, 1990), pp. 746–757
29. C. Scholl, B. Becker, Checking equivalence for partial implementations, in *Proceedings of DAC*, Las Vegas, NV (ACM, New York, 2001), pp. 238–243
30. C.-J.H. Seger, R.E. Bryant, Formal verification by symbolic evaluation of partially-ordered trajectories. Formal Methods Syst. Des. **6**(2), 147–189 (1995)
31. C.-J.H. Seger, R.B. Jones, J.W. O'Leary, T.F. Melham, M. Aagaard, C. Barrett, D. Syme, An industrially effective environment for formal hardware verification. IEEE Trans. CAD Integr. Circuits Syst. **24**(9), 1381–1405 (2005)
32. G.S. Tseitin, On the complexity of derivation in propositional calculus, in *Studies in Constructive Mathematics and Mathematical Logic Part 2* (Springer, Berlin, 1970), pp. 115–125
33. R. Wimmer, K. Gitina, J. Nist, C. Scholl, B. Becker, Preprocessing for DQBF, in *Proceedings of SAT*, ed. by M. Heule, S. Weaver. Lecture Notes in Computer Science, Austin, TX, vol. 9340 (Springer, Berlin, 2015), pp. 173–190
34. K. Wimmer, R. Wimmer, C. Scholl, B. Becker, Skolem functions for DQBF, in *Proceedings of ATVA*, Lecture Notes in Computer Science, Chiba, vol. 9938 (Springer, Berlin, 2016), pp. 395–411
35. K. Wimmer, R. Wimmer, C. Scholl, B. Becker, Skolem functions for DQBF (extended version). Technical Report, FreiDok, Freiburg im Breisgau (2016), https://www.freidok.uni-freiburg.de/data/11130
36. R. Wimmer, S. Reimer, P. Marin, B. Becker, HQSPRE–an effective preprocessor for QBF and DQBF, in *Proceedings of TACAS, Part I*, ed. by A. Legay, T. Margaria. Lecture Notes in Computer Science, Uppsala, vol. 10205 (Springer, Berlin, 2017)

Chapter 8
Progressive Generation of Canonical Irredundant Sums of Products Using a SAT Solver

Ana Petkovska, Alan Mishchenko, David Novo, Muhsen Owaida, and Paolo Ienne

1 Introduction

Minimization of the two-level *Sum Of Products (SOP)* representation is well-studied due to the wide use of SOPs. In the past, research in SOPs was motivated by mapping into *Programmable Logic Arrays (PLAs)*; now SOPs are supported in many tools for logic optimization and are used for multi-level logic synthesis [3, 25], delay optimization [20], test generation [9], but they are also used for fuzzy modelling [10], data compression [1], photonic design automation [5], and in other areas.

These publications show that, contrary to the popular belief, research in SOP minimization and its applications is not outdated. As an example, a recent work uses SOPs for delay optimization in technology independent synthesis and technology

A. Petkovska (✉) • P. Ienne
School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
e-mail: ana.petkovska@epfl.ch; paolo.ienne@epfl.ch

A. Mishchenko
Department of EECS, University of California, Berkeley, Berkeley, CA, USA
e-mail: alanmi@berkeley.edu

D. Novo
French National Centre for Scientific Research (CNRS), University of Montpellier, LIRMM, Montpellier, France
e-mail: david.novo@lirmm.fr

M. Owaida
Eidgenössische Technische Hochschule Zürich (ETHZ), Zürich, Switzerland
e-mail: mohsen.ewaida@inf.ethz.ch

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_8

mapping [20]. In this work, improved quality is achieved by enumerating different SOPs of the local functions of the nodes, factoring them, and finding circuit structures balanced for delay.

Another important application of SOP minimization, which is targeted and used as case-study in this paper, is *global circuit restructuring*. If a multi-level circuit structure for a Boolean function is not available, or if the circuit structure is with poor quality, then a new circuit structure with desirable properties, such as low area, short delay, good testability, or improved implicativity (if the circuit represents constraints in a SAT solver), should be derived. The best known and widely used method for global circuit restructuring is computing SOPs of the output functions in terms of inputs, factoring the multi-output SOPs and deriving a new circuit structure from the shared factored form. The main drawback of this method is the lack of scalability of the algorithm for SOP generation and minimization.

Starting with the Quine-McCluskey algorithm [16], many algorithms and heuristics for SOP generation and minimization have been developed. Prior research falls into two broad categories: BDD-based algorithms and ESPRESSO-style algorithms.

To generate an SOP for a given Boolean function, techniques based on *Binary Decision Diagrams (BDDs)*, such as those of Minato-Moreale [18] and SCHERZO [6, 7], first build a BDD or a *Zero-suppressed Decision Diagram (ZDD)*, then minimize the BDD/ZDD size by using some heuristic approach to obtain a smaller SOP, and finally convert the BDD/ZDD to an SOP. If building a BDD is feasible, then an SOP, even a suboptimal one, can be generated. However, for some circuits, the BDD construction suffers from the BDD memory explosion problem—the BDD size is exponential in the number of input variables—and thus, using BDDs is often impractical. Additional drawback is that BDDs are incompatible with incremental applications since they require building a BDD for the complete circuit before converting it to an SOP. Despite these issues, to our knowledge, the BDD-based method for SOP generation and minimization is used in most of the industrial tools, and therefore scalability improvements of it are highly desirable.

On the other hand, the ESPRESSO-style algorithms are inspired by the first version of ESPRESSO [3]. For example, the logic minimizer ESPRESSO-MV [26] is a faster and more efficient version of ESPRESSO. But, although these techniques avoid the memory explosion problem inherent in the use of BDDs, they still incur impractical runtimes for large Boolean functions and only minimize existing SOPs.

Alternatively, recent progress in the performance of *Boolean satisfiability (SAT) solvers* enabled using SAT in various domains of logic synthesis and verification despite their worst-case exponential runtime. Thus, it has become a trend to replace BDD-based methods with SAT-based ones. For example, this was done for model checking [17], functional dependency [11], functional decomposition [14, 15], and logic don't-care-based optimization [19]. Existing methods for SOP generation using SAT solvers are based on enumeration of satisfying assignments [21]. On the other hand, Sapra et al. [27] proposed using a SAT solver to implement part of ESPRESSO's procedures for SOP minimization in order to speed them up. But, since they largely follow the traditional ESPRESSO style of SOP minimization, they operate only on existing SOPs and do not consider generating a new SOP from a multi-level representation of the Boolean function. Moreover, its runtime and end results significantly depend on the SOP received as input. To the best of our knowledge, there is still no complete SAT-based method for SOP generation similar to the *Irredundant Sum-of-Product (ISOP)* algorithm for incompletely-specified functions using BDDs [18].

Accordingly, the main contribution of this paper is to propose a new engine for SOP generation and minimization that is completely based on SAT solvers. Our method generates the SOP progressively, building it cube by cube. We guarantee that the generated SOPs are *irredundant*, meaning that no literal and no cube can be deleted without changing the function. As we show in the result section, our algorithm generates SOPs with the size similar to that of the BDD-based method [18]. Interestingly, for some circuits, we generate smaller SOPs (up to 10%), which is useful in practical applications. For example, when a multi-level description of the circuit is built using an SOP produced by the proposed SAT-based method, the area-delay product of the resulting circuit, assuming unit-area and unit-delay model, often decreases (up to 27%), compared to global restructuring using BDDs.

Two main features characterize our SAT-based SOP generation and make it desirable in various domains.

First, we generate an SOP *progressively*, unlike BDD-based methods that attempt to construct a complete SOP at once. The progressive computation allows generation of *a partial SOP* for circuits whose complete SOP cannot be computed given the resource limits. The partial SOPs can be exploited by other applications that do not require the complete circuit functionality, but work with partially defined functions [4, 30]. Moreover, for circuits with large SOPs, the progressive generation allows us to predict whether it is feasible to build an SOP for a circuit, and to check if the SOP size is within the limits of the methods that are going to use it. For this, at any moment, we can retrieve the number of outputs for which the SOP is already computed, as well as the finished SOP portion of the currently processed output. We can also easily compute an estimate or a lower-bound of the percentage of covered minterms, considering uniform distribution of minterms in the space or considering the size of the truth table, respectively. In contrast, the termination time and the quality of results of the BDD-based methods are unpredictable since the complete BDD has to be built before converting it to an SOP.

Second, counter-intuitive as it may sound, we show that the SAT-based computation can generate *canonical* SOPs. To this end, we combine (1) an algorithm that, under a given variable order, generates consecutive SAT assignments in lexicographic order [24], considering each assignment as integer value, and (2) a deterministic algorithm that expands the received assignments into cubes. For a given function and a variable order, the assignments (i.e., the minterms) are always generated in the same order, and each assignment always results in the same cube. Thus, the resulting SOP is canonical—it is unique and independent of the input implementation of the function. The canonical nature of the resulting SOPs can be useful in those domains where previously only BDDs could be used. For example, applications as constraint solving [31] and random assignment generation [22] can benefit from the canonicity if we iterate repeated generation of random valuation of inputs and get the closest SAT assignment, as it is done in the proposed canonical SOP generation method. Also, the canonicity brings regularity in the SOPs, and thus the results after using algorithms for factoring [25] are in some cases better.

In the rest of the paper, we focus on completely-specified functions, but the given SAT-based formulation works for incompletely-specified functions without any changes. Indeed, after extracting the first cube and blocking it in the on-set of the function, the rest of the computation is performed for the incompletely-specified functions, even if the initial function was completely specified.

The rest of the paper is organized as follows. Section 2 gives background on Boolean functions, the SOP representation, and the satisfiability problem. Next, we describe our algorithm for SAT-based progressive generation of irredundant SOPs in Sect. 3. Section 4 gives our experimental setup and discusses the experimental results. Finally, we conclude and present ideas for future work in Sect. 5.

2 Background Information

In this section, we define the terminology associated with Boolean functions and the SOP representation, as well as with the satisfiability problem.

2.1 Boolean Functions

For a variable v, a *positive literal* represents the variable v, while the *negative literal* represents its negation $\bar{v}$. A *cube*, or a product, c, is a Boolean product (AND, $\cdot$) of literals, $c = l_1 \cdot \cdots \cdot l_k$. If a variable is not represented by a negative or a positive literal in a cube, then it is represented by a *don't-care* (−), meaning that it can take both values 0 and 1. A cube with i don't-cares covers 2^i minterms. A *minterm* is the smallest cube in which every variable is represented by either a negative or a positive literal.

Let $f(X) : B^n \rightarrow \{0, 1, -\}$, $B \in \{0, 1\}$, be an *incompletely-specified Boolean function* of n variables $X = \{x_1, \ldots, x_n\}$. The terms function and circuit are used interchangeably in this paper. The *support set* of f is the subset of variables that determine the output value of the function f. The set of minterms for which f evaluates to 1 defines the *on-set* of f. Similarly, the minterms for which f evaluates to 0 and don't-care define the *off-set* and the *don't-care-set*, respectively. In a multi-output function $F = \{f_1, \ldots f_m\}$, each output f_i, $1 \leq i \leq m$, has its own support set, on-set, off-set, and don't-care-set associated with it.

For simplicity, we define the following terms for single-output functions, although our algorithm can handle multi-output functions. Any Boolean function can be represented as a two-level *sum of products (SOP)*, which is a Boolean sum (OR, +) of cubes, $S = c_1 + \cdots + c_k$. Assume that a Boolean function f is represented as an SOP. A cube is *prime*, if no literal can be removed from the cube without changing the value that the cube implies for f. A cube that is not prime can be *expanded* by substituting at least one literal with a don't-care. The SOP is *irredundant* if each cube is prime and no cube can be deleted without changing the function.

A *canonical representation* is a unique representation for a function under certain conditions. For example, given a Boolean function and a fixed input variable order, a canonical SOP is an SOP independent of the original representation of the function given to the SAT solver, of the CNF algorithm, and of the used SAT solver. In a similar way, BDDs generate a canonical SOP that only depends on an input variable order [18].

2.2 Boolean Satisfiability

A disjunction (OR, $+$) of literals forms a *clause*, $t = l_1 + \cdots + l_k$. A *propositional formula* is a logic expression defined over variables that take values in the set $\{0, 1\}$. To solve a SAT problem, a propositional formula is converted into its *Conjunctive Normal Form (CNF)* as a conjunction (AND, $\cdot$) of clauses, $F = t_1 \cdot \cdots \cdot t_k$. Algorithms such as the Tseitin transformation [29] convert a Boolean function into a set of CNF clauses.

A *satisfiability (SAT) problem* is a decision problem that takes a propositional formula in CNF form and returns that the formula is *satisfiable* (*SAT*) if there is an assignment of the variables from the formula for which the CNF evaluates to 1. Otherwise, the propositional formula is *unsatisfiable* (*UNSAT*). A program that solves SAT problems is called a *SAT solver*. SAT solvers provide a *satisfying assignment* when the problem is satisfiable.

Modern SAT solvers can determine the satisfiability of a problem under given assumptions. *Assumptions* are propositions that are given as input to the SAT solver for a specific single invocation of the SAT solver and have to be satisfied for the problem to be SAT.

Example 1 For the function $f(x_1, x_2, x_3) = (x_1 + x_2)\bar{x}_3$, which is satisfiable for the following assignments of the inputs $\{(0, 1, 0), (1, 0, 0), (1, 1, 0)\}$, a SAT solver without assumptions can return any of the given assignments. But, if we give as input to the SAT solver the assumption $x_1 = 1$, then it returns either $(1, 0, 0)$ or $(1, 1, 0)$, because those two assignments satisfy the given assumption.

A *lexicographic satisfiability (LEXSAT) problem* is a SAT problem that takes a propositional formula in CNF form and, given a variable order, returns a satisfying variable assignment whose integer value under the given variable order is minimum (maximum) among all satisfying assignments. The returned satisfying assignment is called a *LEXSAT assignment*. If the formula has no satisfying assignments, LEXSAT proves it unsatisfiable. There are several solutions for the LEXSAT problem [12, 23, 24]. For our work, we use an efficient algorithm for generating consecutive LEXSAT assignments [24].

Example 2 For the function $f(x_1, x_2, x_3)$ from Example 1, LEXSAT returns either the lexicographically smallest assignment (0, 1, 0) or the lexicographically greatest assignment $(1, 1, 0)$, depending on the user preference.

3 SAT-Based SOP Generation

In this section, we describe our SAT-based algorithm that progressively generates an irredundant SOP for a single-output function. For multi-output circuits, each output is treated separately. In this paper, we focus on completely-specified functions, but the algorithm can be easily used for incompletely-specified functions by providing both the on-set and off-set as input to the algorithm. In the case of a completely-specified function one of them is derived by complementing the other.

The presented algorithm iteratively generates minterms, expands them into prime cubes, and adds these cubes to the SOP. The SAT-based heuristics for minterm generation and cube expansion are described in Sects. 3.1 and 3.2, respectively. Finally, to guarantee that the resulting SOP is irredundant, it is post-processed to remove redundant cubes, as described in Sect. 3.3. Additionally, Sect. 3.4 describes several techniques that reduce the runtime.

The algorithm can be implemented with one SAT solver parameterized to store both on-set and off-set. Alternatively, it can use two solvers, one for on-set and one for off-set. In our implementation of the algorithm, we use four different SAT solvers: for both on-set and off-set, one is used to generate satisfying assignments, the other to expand assignments to cubes. By employing four solvers, we ensure that assignment generation and expansion do not interact with each other during the SOP computation.

The procedures described in the following subsections assume that we are generating the on-set SOP. The same procedures are used to generate the off-set SOP, by substituting the on-set SAT solver with an off-set SAT solver and vice versa.

3.1 Generation of Minterms

In order to generate minterms for the on-set of a function f by using a SAT solver, we initialize a SAT solver with the CNF of f. Then, to discard the trivial case when the function has a constant on-set, we solve the SAT problem by asserting that $f = 0$. If the problem is UNSAT, then f is a constant, and we return an SOP with one constant cube. Otherwise, if the problem is SAT, we continue with the following methods for minterm generation. Figure 8.1 shows the flowchart of these methods, as well as their connection with the other methods of the SOP generation algorithm.

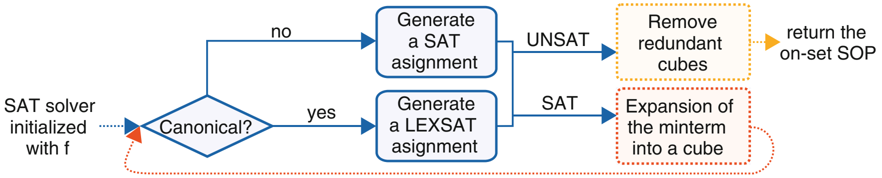


Fig. 8.1 Flowchart of the algorithm for minterm generation. Minterms are generated either as SAT or LEXSAT assignments. If the problem is SAT, the generated minterm is passed to the cube expansion algorithm to generate a cube that would cover the minterm. Once all minterms are covered by the generated cubes, the SAT problem becomes UNSAT, and the SOP is returned after removing the redundant cubes

Generation of a Non-canonical SOP When the problem is SAT, an assignment for the inputs is returned for which the function evaluates to 1. From the assignment, we can generate a minterm for the function f in which the variables assigned to 0 and 1 are represented with the negative and positive literal, respectively. For example, for a function $f(x, y, z)$, the assignment $(1, 1, 0)$ implies the minterm $xy\bar{z}$. Once a minterm is obtained, we expand it into a cube using the heuristic procedure from Sect. 3.2. Next, we add the cube with its literals complemented to the SAT solver as a *blocking clause*, which is an additional clause that blocks known solutions of the SAT problem. This allows to generate a new minterm that is not covered by any of the previously generated cubes. While the problem is SAT, we iteratively obtain a minterm, expand it to a cube, and add the cube to both the SAT solver and the SOP. The unsatisfiability of the problem indicates that the generated SOP is complete and covers all on-set minterms.

Generation of a Canonical SOP Generating minterms from satisfying assignments received from a SAT solver does not guarantee canonicity, since SAT solvers return minterms in a non-deterministic order that depends on the design of the solver and the CNF generated for the function. Thus, to ensure canonicity, we iteratively use a binary search-based LEXSAT algorithm, called `BINARY` [24], that generates minterms in a lexicographic order that is unique for a given variable order. The algorithm `BINARY` receives as input a potential assignment, which is the lexicographically smallest assignment that might be satisfying, that is either the last generated minterm or, initially, an assignment with all 0s. Then, `BINARY` tries to verify and fix the assignment of each variable defined with the potential assignment starting from the leftmost variables and moving to right. We also use the proposed methods for runtime improvement [24]: skip verifying the leading 1s, correcting the initial potential assignment, and profiling the success of the first SAT call. Similarly to the non-canonical SOPs, once we obtain a minterm, we expand it into a cube and add it to the SAT solver as a blocking clause.

Example 3 For example, assume that for the function $f(x_1, \ldots, x_8)$, the last generated minterm $(1, 1, 0, 0, 0, 0, 0, 1)$ is received as an initial potential assignment. Since this minterm is covered by the last cube, this assignment is not satisfying,

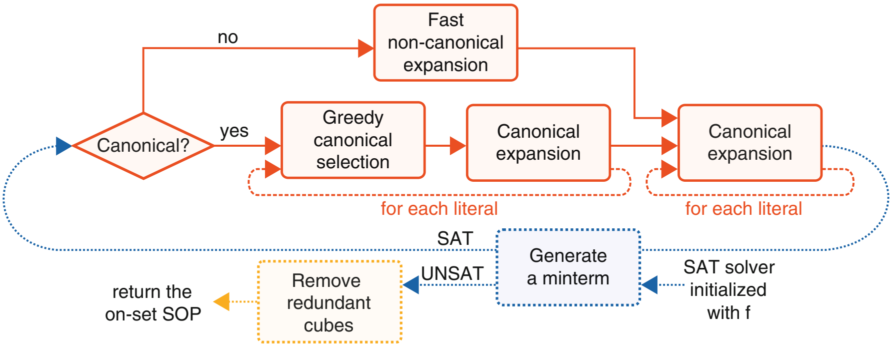


Fig. 8.2 Flowchart of the algorithm for expansion of minterms into cubes. The algorithm for canonical expansion ensures that all generated cubes are prime. After a cube is generated, it is added as a blocking clause to the SAT solver used for minterm generation, and another minterm is generated

so we can increase its value for 1 to get the smallest assignment that might be satisfying $(1, 1, 0, 0, 0, 0, 1, 0)$. Next, we can skip verifying the assignments $x_1 = 1$ and $x_2 = 1$, because the next lexicographically smallest assignment has to start with the same leading 1s. Thus, we should only check the assignments for x_i, for $3 \leq i \leq 8$. Due to using binary search, with the first SAT call we assume half of the unfixed assignments, and we give to the on-set SAT solver the assumptions $(x_1, \ldots, x_5) = (1, 1, 0, 0, 0)$. Assume that the problem was satisfiable and the SAT solver returned the assignment $(1, 1, 0, 0, 0, 0, 1, 1)$. This assignment proves that an on-set minterm with the assumed values exists, but moreover we can learn that the assignments from the potential minterm $x_6 = 0$ and $x_7 = 1$ are correct. Next, to check if the assignment for the last input x_8 can be set to 0, we call the SAT solver with the assumptions $(x_1, \ldots, x_8) = (1, 1, 0, 0, 0, 0, 1, 0)$. If it returns SAT, we return the potential assignment as a minterm since all assignments are verified and fixed. Otherwise, we flip x_8 to 1 to increase the potential assignment before returning it.

3.2 Expansion of Minterms into Cubes

In this subsection, we describe our SAT-based procedure that receives a minterm and transforms it into a prime cube by iteratively removing literals (i.e., substituting them with don't-cares). For the on-set SOP, a literal can be removed, if after its removal all minterms covered by the cube do not overlap with the off-set. Figure 8.2 shows a flowchart of the algorithm.

Canonical Expansion to Prime Cubes The following deterministic algorithm expands a minterm into a cube by ensuring that, after removing each literal, the

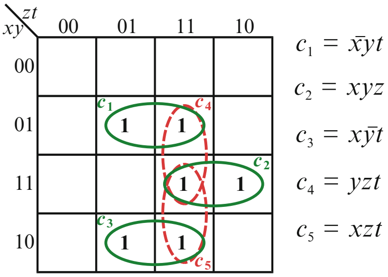


Fig. 8.3 A Karnaugh map for the Boolean function $f(x, y, z, t) = \bar{x}yt + xyz + x\bar{y}t$ with its prime cubes c_i, where $1 \le i \le 5$. The cubes c_1, c_2, and c_3 are essential and they compose the minimum SOP of f

cube is covering only on-set minterms. Since the literals are removed always in the same order, which can be specified by the user, the algorithm is deterministic and produces canonical cubes if the given minterms are canonical. Thus, to remove a literal, first, we assume that the literal is removed from the cube, and an off-set SAT solver is run with assumptions for the remaining literals of the cube. If the problem is UNSAT, then no minterm covered by the cube belongs to the off-set, so we can extend the cube by removing this literal. On the other hand, if the problem is SAT, we cannot extend the cube, since the SAT solver found an off-set minterm that is covered by the extended cube.

Example 4 Assume that for the function in Fig. 8.3, we received the minterm $\bar{x}y\bar{z}t$. To remove the literal $\bar{x}$, we would call the off-set SAT solver with the assumptions $(y, z, t) = (1, 0, 1)$. The SAT solver would return SAT, which means that $\bar{x}$ cannot be removed, because the cube $y\bar{z}t$ is covering the off-set minterm $xy\bar{z}t$. However, if we try to remove the literal $\bar{z}$ by calling the SAT solver with the assumptions $(x, y, t) = (0, 1, 1)$, then we would receive UNSAT because there are no off-set minterms that satisfy these assumptions, so $\bar{z}$ can be removed to obtain the on-set cube c_1.

Greedy Canonical Cube Expansion To minimize the overlapping of cubes, we propose to remove literals in two rounds. In the first round, they are removed greedily, after ensuring that multiple on-set minterms are covered by expanding each literal.

Example 5 Assume that for the function in Fig. 8.3, the cube c_1 was computed and added to the on-set SAT solver as a blocking clause. Also, assume that as a second minterm $xyzt$ is generated, which can be extended by removing one of the literals x, y or t. If we remove x, we will obtain the cube c_4 that covers only one additional minterm with respect to the existing cube c_1, but if we remove y or t, we will obtain c_2 or c_5, respectively, each of which covers two yet uncovered minterms.

In Example 5, our expansion procedure skips the opportunity to remove the literal x, and tries to expand other literals if possible. This greedy selection of literals decides to candidate a literal l_i for removal, if by removing it, the expanded cube covers more than one new minterm. To check if this condition is satisfied, we flip l_i and provide it, along with the remaining literals of the cube, as assumptions to an on-set SAT solver in which the already generated cubes are added as blocking clauses.

If the problem is UNSAT, then we skip removing it temporarily. Otherwise, if the problem is SAT, then we consider this literal for removal since by removing it we cover more than one uncovered minterm. Once a literal is a candidate for removal, we run the algorithm for canonical expansion described above to ensure that it can be removed.

However, in this first round, we might skip some opportunities for expansion. Thus, in the second round, for each skipped literal, we execute the algorithm for canonical expansion. This guarantees that, after the second round, no literal can be further removed, which means that the cube is prime. Since, we always try to remove the literals in the same user specified order, this method generates a canonical SOP.

Fast Non-canonical Expansion If generating a canonical SOP is not required, we can substitute the first round of expansion with a faster method to improve runtime: If in an off-set SAT solver we assume the values from the received on-set minterm, the problem is UNSAT and the SAT solver returns the set of literals used to prove unsatisfiability (procedure "analyse_final" in MiniSAT [8]). Since the returned literals are sufficient to prove unsatisfiability in an off-set SAT solver, they construct a cube that covers only on-set minterms, and we can remove literals that are not returned by the SAT solver. However, the set of remaining literals is not always minimal, and thus we run additionally the algorithm for canonical expansion as a second round to obtain a prime cube.

3.3 Removing Redundant Cubes

The cubes expanded with the methods from Sect. 3.2 are prime by construction. However, by progressively adding cubes to the SAT solver, as described in Sect. 3.1, we ensure that each cube is irredundant with respect to the preceding cubes, but not with respect to the whole set of cubes.

Example 6 For the function f from Fig. 8.3, assume that the cubes c_1, c_5, c_2, and c_3 are generated in the given order. The cube c_5 is irredundant with respect to c_1, since it additionally covers the minterms $xyzt$ and $x\bar{y}zt$, but it is contained in the union of c_2 and c_3.

In order to produce an irredundant SOP, after generating all cubes, we iterate through the cubes to detect and remove redundant ones. First, we initialize a new SAT solver with clauses for all generated cubes and we assume that all cubes are required. Then, by using the assumption mechanism, for each cube c_i, we check if there is an assignment for which c_i evaluates to 1 while all the other irredundant cubes evaluate to 0. If the problem is SAT, the cube is irredundant and the SAT solver returns an assignment that corresponds to a minterm which is covered only by c_i. Otherwise, if the problem is UNSAT, then the cube is redundant, and thus it is removed from the SOP and is excluded when checking the redundancy of the following cubes. Since we always try to remove cubes in the order in which they were generated, this method is deterministic and maintains canonicity when canonical SOPs are generated.

Example 7 Considering the cubes from Example 6, to check whether c_3 is redundant, we set $c_3 = 1$ by assuming the values $x = 1, y = 0$ and $t = 1$. For the assumed values, the other cubes evaluate to $c_1 = 0$, $c_2 = 0$ and $c_5 = z$. Setting $z = 0$ leads to $c_5 = 0$. Thus, the problem is SAT and c_3 is irredundant. The returned assignment $(x, y, z, t) = (1, 0, 0, 1)$ defines a minterm $x\bar{y}\bar{z}t$ that is covered only by c_3.

3.4 Improving the Runtime

In this subsection, we present four techniques that improve the runtime of the algorithm by allowing early termination and by treating some special cases.

Simultaneous On-Set and Off-Set Generation Often, the SOP of the on-set and off-set differ in size. For example, a three-input function implementing an AND gate, $f(x, y, z) = xyz$, has an on-set SOP, $f = S_{\mathrm{on}} = xyz$ with size 1, and an off-set SOP, $\bar{f} = S_{\mathtt{off}} = \bar{x} + \bar{y} + \bar{z}$ with size 3. Since we want to use the set with a smaller SOP, we simultaneously generate two SOPs, for both the on-set and off-set, by generating one cube at a time from each set, and we stop the generation when one SOP is complete. This way, if one of set is much smaller than the other, we can avoid the situation when the larger set of cubes has to be completely generated, before the smaller set is discovered.

Prioritizing Outputs with Large SOPs Before generating SOPs for each output, we propose to sort outputs by size of their input supports. The outputs with larger supports are processed first since it is more likely that the SOP generation for these outputs will exceed resource limits, so we can determine if we should terminate the computation earlier.

Isomorphic Circuits To benefit from the structure sharing among the circuit outputs, we implemented a method that decreases the runtime by detecting isomorphic outputs. For this, first, we divide the outputs into isomorphic classes. Two outputs are *isomorphic* and belong to the same class, if they implement an identical function using different inputs. Then, for each class, we generate an SOP only for one output, which is the class representative, and duplicate it for the others. In Sect. 4.2, we show that this allows effective generation of an SOP only for 16.5% of all combinatorial outputs and has a big influence on scalability.

CNF Sharing Generating a CNF for each output is time consuming. Thus, to benefit from the logic sharing among the outputs, we can optionally share one CNF, which corresponds to the complete circuit. For this, we generate the CNF of the circuit, and then, for each output, we initialize the SAT solver only with the part of the CNF for the corresponding output. Besides improving the runtime, as Table 8.1 shows, this option sometimes leads to better results in terms of area-delay product after global restructuring.

Table 8.1 Number of benchmarks (out of the 71 used benchmarks) for which activating or deactivating an option for SATCLP results in the smallest SOP in terms of number of cubes (columns under "#Cubes") or the best area-delay product (columns under "Area·Delay"). If for one benchmark, an identical best result is obtained both when the option is activated and deactivated, then we count it as a tie

	#Cubes			Area·Delay		
	No	Yes	Tie	No	Yes	Tie
Canonical	7	34	30	28	26	17
Shared CNF	43	1	27	40	13	18
Order PI	45	8	18	57	11	3
Reverse PI	20	15	36	28	21	22

Exploiting Parallelism There are several opportunities where computations are independent and can be parallelized. First, the computation of the on-set and off-set SOPs can be executed in parallel. Since now we compute sequentially one cube for each SOP interchangeably, it is expected that this would decrease the runtime by 2×. Second, instead of computing the SOP of each output one after the other, we can also compute each of them in parallel. Finally, for one SOP, we can compute cubes in parallel by generating minterms from different parts of the Boolean space. However, in this paper, all computations are done sequentially. Analyzing and exploiting the effect of parallelism is left for future work.

4 Experimental Results

In this section, we describe our experimental setup and compare the proposed SAT-based algorithm with a state-of-the-art BDD-based method.

4.1 Experimental Setup

We implemented the SAT-based algorithm described in Sect. 3 as a new command *satclp* in ABC [2]. ABC is an open-source tool for logic synthesis, technology mapping, and formal verification of logic circuits. ABC features an integrated SAT solver based on an early version of MiniSAT [8] that supports incremental SAT solving. Furthermore, ABC provides an implementation of the BDD-based method for SOP generation, namely the BDD construction for a multi-level circuit (command *collapse*) and the BDD-based ISOP computation [18] (command *sop*). For convenience, in this section, we refer to the SAT-based and BDD-based methods as SATCLP and BDDCLP, respectively. Finally, ABC allows us to analyze the area-delay results when the generated SOPs are used to build a new multi-level circuit structure. A multi-level network is generated using the *fx* command [25]. The network is next converted into an *And-Inverter Graph (AIG)* (command *strash*), which is an internal representation of ABC, and optimized with the *dc2* command. The area and delay of the resulting AIGs are compared for different SOP generation methods.

To evaluate our algorithm, we use the ISCAS'89 benchmarks, a set of large MCNC benchmarks, a set of nine logic tables from the instruction decoder unit [28] denoted as LT-DEC, and a set of proprietary industrial benchmarks. The LT-DEC suite is well suited to demonstrate the factoring gains as circuit size increases [13]. The names of the LT-DEC benchmarks are given in the form "[$N_{\mathtt{PI}}$]/[$N_{\mathtt{PO}}$]", where $N_{\mathtt{PI}}$ is the number of primary inputs and $N_{\mathtt{PO}}$ is the number of primary outputs. For the main experiments, we discard benchmarks for which the SOP size exceeds the built-in resource limits of the used commands, and thus, we use 30 (out of 32) benchmarks from the ISCAS'89 set, 15 (out of 20) benchmarks from the MCNC set, and 17 (out of 18) industrial benchmarks. With the discarded benchmarks, we demonstrate the generation of partial SOPs.

4.2 SAT-Based vs. BDD-Based SOP Generation

To analyze the performance of the algorithm presented in Sect. 3, we run both `SATCLP` and `BDDCLP` available in ABC. In this section, we present the results of these experiments.

Although the command *collapse* dynamically finds a good variable order for the BDD, changing the initial order of the primary inputs results in a different BDD structure, which leads to a different SOP. Thus, to obtain a good SOP, we generate five SOPs for `BDDCLP` by using five different initial orders of the primary inputs. Similarly, `SATCLP` generates different SOPs for different orders of the primary inputs, which define the order of removing literals from the cubes. We either use the pre-defined order from the benchmark file or order the inputs based on their number of fanouts (option "Order PI"), which currently works only for the combinational benchmarks. We can also, optionally, reverse the selected variable order (option "Reverse PI"). Moreover, we can enable generation of canonical SOPs (option "Canonical"), and for non-canonical SOPs we can enable generating one CNF for all outputs as described in Sect. 3.4 (option "Shared CNF"). Thus, by changing these four options, we generate 12 SOPs using `SATCLP`.

Generating multiple SOPs with each method results in SOPs that differ in size, where the SOP size is equal to the number of cubes that constitute the SOP. Figure 8.4 shows and compares the benchmarks for which the size of the smallest SOP generated by each method is different. Although `SATCLP` most often generates SOPs with almost the same size as those generated by `BDDCLP`, for some benchmarks it generates smaller SOPs (up to 10%). Since the results for `SATCLP` are obtained using several different options, Table 8.1 shows, under "#Cubes", the number of benchmarks for which the smallest SOP is generated when a given option is deactivated or activated. We can notice that, for 34 benchmarks we get exclusively

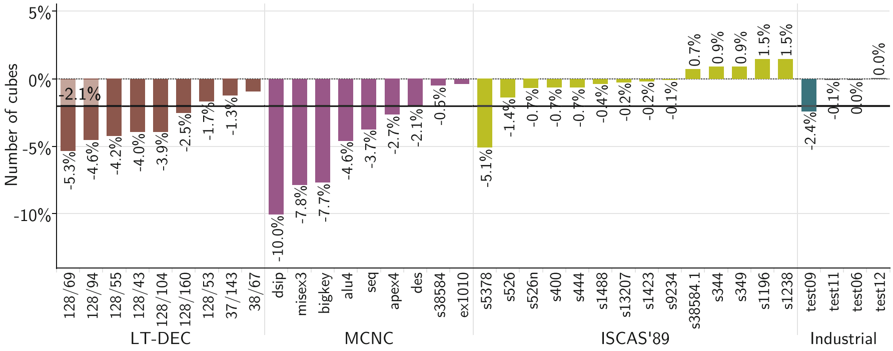


Fig. 8.4 Size of the smallest SOPs generated by SATCLP compared to the smallest SOP generated by BDDCLP. Only the benchmarks for which the SOP size differs are shown. The *gray line* shows that, on average, SATCLP decreases the SOP size by 2.1%

Table 8.2 Comparison of the number of combinational outputs, which are primary outputs and latch inputs, in the used benchmarks and the number of isomorphic classes, which is equal to the number of calls of the SAT-based algorithm for SOP generation

Set	Number of benchmarks	Combinational outputs	Isomorphic classes	Ratio(%)
LT-DEC	9	788	686	87.1
MCNC	15	3024	1435	47.5
ISCAS'89	30	5753	1709	29.7
Industrial	17	64,267	8356	13.0
Total	71	73,832	12,186	16.5

smaller SOP when generating canonical SOPs, and only for 7 benchmarks the non-canonical SOPs are smaller. Similarly, the SOP size increases for about 60% of the benchmarks if the CNF is shared or if the inputs are ordered by their number of fanouts.

Next, we compare the algorithms' runtime. The reported runtime is average over three runs of the algorithm for SOP generation. For BDDCLP, we report the time required to execute the commands *collapse* and *sop*. For SATCLP, we report the time taken by our command *satclp*, which includes the time to generate isomorphic outputs, derive CNF, and initialize SAT solver instances, as well as the time for all SAT calls for minterm generation, cube expansion, and removing redundant cubes.

In terms of scalability, as Table 8.2 shows, the idea of filtering out structurally isomorphic outputs presented in Sect. 3.4 allows computing an SOP only for 16.5% of the combinational outputs, one for each isomorphic class, while for the other outputs we duplicate the generated SOP of the class representative. This reduces the runtime of our algorithm SATCLP, and for benchmarks rich in isomorphic outputs, the proposed method is significantly faster than BDDCLP. For example, from the public benchmarks, the maximum speedup is achieved for the benchmark s35932 from the ISCAS'89 set, for which we generate SOPs only for 10 out of 2048 combinational outputs and thus, on average, SATCLP requires 0.10 s, while BDDCLP requires 1.57 s. However, on average, our SATCLP is 7.5× slower than BDDCLP for the public benchmarks. We have observed that the functions for expanding minterms into cubes are the bottleneck. For example, for the LT-DEC benchmarks, on average, 85% of the runtime is spent in this operation, while 8% is spent on minterm generation, 2% on removing redundant cubes, and 5% on other operations, such as dividing the outputs into classes, generating CNF, initializing SAT solver instances, etc.

On the other hand, Table 8.3 shows runtime results for a suite of proprietary industrial benchmarks. We can see that SATCLP is often faster than BDDCLP, especially for the benchmarks that have many isomorphic outputs, and is definitely more scalable, that is, it completes on some test-cases, for which BDDCLP fails. For example, for the non-canonical SOPs, on average, SATCLP decreases the runtime of SOP generation by 45.9%. For canonical SOPs, although SATCLP is 5.2× slower than its non-canonical version and 2.9× slower than BDDCLP, it successfully generates SOPs for 5 benchmarks, for which BDDCLP fails.

We believe that the increased scalability of SATCLP is largely due to the fact that most of the industrial testcases have hundreds of inputs and outputs, which makes constructing global BDDs in the same manager problematic for all outputs at once. The algorithm SATCLP does not suffer from this limitation, because it computes the SOPs for one output at a time. It can be argued that the BDD-based computation can also be performed on the per-output basis. However, in this case, the BDD manager will inevitably find different variable orders for different outputs, which will increase the size of the resulting multi-level circuits when these SOPs are factored. In fact, factoring benefits from computing SOP with the same variable order that facilitates creating similar combinations of literals in different cubes, which in turn helps improve the quality of shared divisor extraction and factoring.

Finally, since we generate cubes progressively, unlike BDDCLP, we can build partial SOPs even for large circuits, and these can be used for incremental applications. Figure 8.5 shows the number of cubes composing the partial non-canonical SOPs for which a time limit for the runtime is set to t seconds, where t is an integer value such that $1 \leq t \leq 10$. For functions with larger supports, we usually generate less cubes because more time is required for cube expansion. Only for the benchmark test14 we are not able to generate any cube in the first 6 s due to the large support set of the first processed output, which depends on 6246 inputs. For the other benchmarks, we generate thousands of cubes in just a few seconds. In this experiment, we are still generating both the on-set and the off-set SOP at the same time. However, in the incremental applications, we can generate just one of them, which would increase the number of generated cubes for a given time limit.

Table 8.3 Runtime results for the combinational industrial benchmarks when SOPs are generated with BDDCLP and SATCLP

				Runtime (s)		
			Isomorphic classes		SATCLP	
	PIs	POs		BDDCLP	Non-canonical	Canonical
test01	2513	2377	2083	31.14	165.99	1658.92
test02	3236	3202	3146	–	32.46	112.15
test03	1542	514	113	10.64	12.74	70.79
test04	37,397	292	155	144.57	15.01	197.71
test05	1178	606	95	–	141.85	748.81
test06	1488	1446	580	4.24	31.50	137.74
test07	8087	335	270	152.42	17.91	68.31
test08	438	512	432	3.96	17.34	84.67
test09	870	1636	792	2.36	18.17	125.19
test10	2376	1233	314	100.83	10.55	46.88
test11	3875	3274	138	14.49	2.49	7.95
test12	4626	3708	112	10.29	1.59	3.17
test13	1110	1040	74	50.86	1.30	9.29
test14	8514	1323	890	–	–	–
test15	47,356	4136	21	–	0.21	0.26
test16	58,382	18,433	9	–	0.63	0.28
test17	68,620	17,411	19	–	0.64	0.33
test18	36,900	4112	3	603.86	277.08	42,292.50
Average				1.00	0.54	2.88

The columns “PIs” and “POs” give the number of primary inputs and outputs, respectively. A dash (–) denotes that the method fails to compute an SOP. Highlighted are the cases when SATCLP outperforms BDDCLP

4.3 Case-Study: SAT-Based SOPs for Generation of Multi-level Implementation

As explained in Sect. 4.2, we generate several SOPs with each method. The different SOPs result in multi-level networks with different area and delay. As Fig. 8.6 shows, for most benchmarks, our algorithm obtains Pareto-optimal solutions, compared to BDDCLP. To obtain these results, we isolate the best circuit implementations in terms of area-delay product as derived by each method. Table 8.1, with the columns “Area·Delay”, shows the number of benchmarks with the smallest area-delay product generated when a given option was deactivated and activated. For 26 benchmarks we generate a circuit structure with smaller area-delay product when

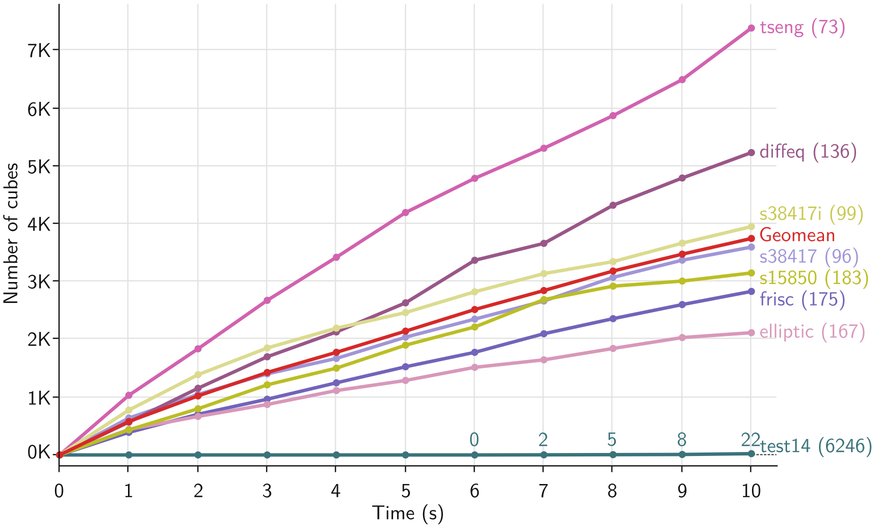


Fig. 8.5 The number of generated cubes for a partial SOP when the time limit is set between 1 and 10 s. The number of generated cubes depends on the size of the support set of the output with largest support set, which is given in brackets. For all benchmarks, the generated cubes belong to one output

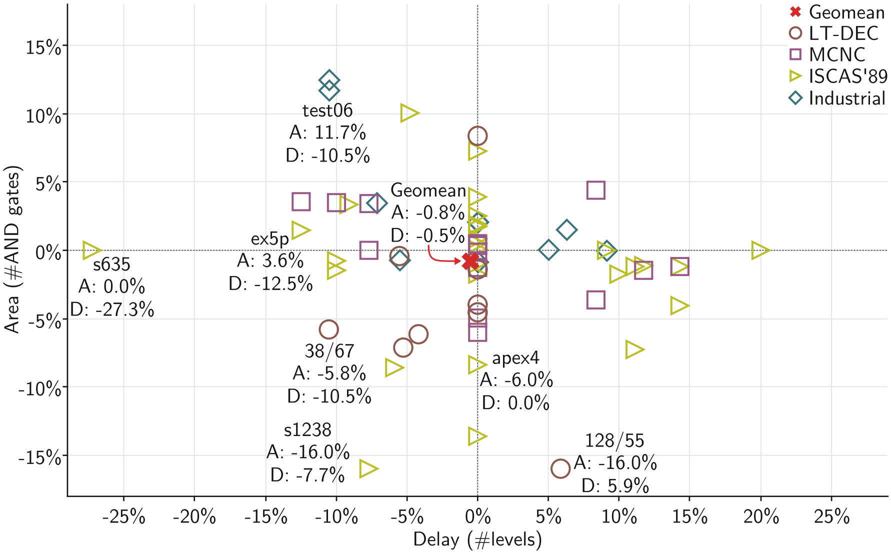


Fig. 8.6 The best results for each benchmark after a multi-level description is built from SOPs generated by our SAT-based algorithm, compared to using a BDD-based SOPs. For most benchmarks, we obtain Pareto optimal solutions

generating canonical SOPs, while for 28 benchmarks the non-canonical SOPs are a better option. Also, for most benchmarks it is best to generate an SOP by using the original ordering of primary inputs used in the benchmark file.

5 Conclusion

In this paper, we present a novel algorithm for progressive generation of irredundant canonical SOPs using heuristics based solely on SAT solving. Besides generating SOPs, the canonicity and the progressive generation make our heuristics desirable in many other areas where minterms or cubes are required, and for which the existing methods are either unscalable or impractical to use.

Regarding the quality of results, we show that for computing a complete SOP, on average, the SAT-based computation is as good as the BDD-based one. Moreover, the multi-level circuit structures derived using the SOPs generated by our approach are often better or Pareto-optimal.

Regarding the runtime, the proposed method is somewhat slower than the BDD-based method for most of the public benchmarks, but it is faster for circuits that are rich in isomorphic outputs. Thus, for the industrial benchmarks, our method is both faster and more scalable, and therefore it is a good candidate for global circuit restructuring at least in that particular industrial setting.

Besides the described opportunities for parallelization, the proposed method can also benefit from the ongoing improvement in modern SAT solvers. For example, recently we explored a new push/pop interface for assumptions used in the incremental SAT solving, which led to additional runtime improvements. As we show, for some circuits the results can improve by changing the variable order in which the cubes are expanded, but a careful study of this problem is required to improve further the quality of results.

In addition to runtime improvements, future work will focus on developing a dedicated SAT-based multi-output SOP computation, which computes cubes that are shared between several outputs. A recent publication [13] indicates that a significant improvement in quality (more than 10%) can be achieved by computing and factoring multi-output SOPs. We are not aware of a practical method for BDD-based multi-output SOP computation, so it is likely that SAT will be the only way to work with multiple outputs. Other directions of future work will include exploring the benefits of the progressive generation of canonical minterms and cubes in different areas. One such area is multi-level logic synthesis where incremental SAT-based decomposition methods can be developed based on partial SOPs computed for the output functions.

Acknowledgements This work is partly supported by NSF/NSA grant “Enhanced equivalence checking in cryptoanalytic applications” at University of California, Berkeley, and partly by SRC contract 2710.001 “SAT-based Methods for Scalable Synthesis and Verification”.

References

1. L. Amarù, P.E. Gaillardon, A. Burg, G. De Micheli, Data compression via logic synthesis, in *Proceedings of the Asia and South Pacific Design Automation Conference*, Yokohama (2014), pp. 628–33
2. Berkeley Logic Synthesis and Verification Group, Berkeley, Calif.: ABC: A system for sequential synthesis and verification, http://www.eecs.berkeley.edu/~alanmi/abc/
3. R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic, Boston, 1984)
4. K. Chang, V. Bertacco, I.L. Markov, A. Mishchenko, Logic synthesis and circuit customization using extensive external don't-cares. ACM Trans. Des. Autom. Electron. Syst. **15**(3), 1–24 (2010)
5. C. Condrat, P. Kalla, S. Blair, Logic synthesis for integrated optics, in *Proceedings of the 21st ACM Great Lakes Symposium on VLSI*, Lausanne (2011), pp. 13–18
6. O. Coudert, Two-level logic minimization: An overview. Integr. VLSI J. **17**(2), 97–140 (1994)
7. O. Coudert, J.C. Madre, H. Fraisse, H. Touati, Implicit prime cover computation: An overview, in *Proceedings of the Synthesis and Simulation Meeting and International Interchange*, Nara (1993)
8. N. Eén, N. Sörensson, An extensible SAT-solver, in *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, vol. 2919 (Springer, Berlin, 2003), pp. 502–18
9. A. Ghosh, S. Devadas, A.R. Newton, Test generation and verification for highly sequential circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **10**(5), 652–67 (1991)
10. A.F. Gobi, W. Pedrycz, Fuzzy modelling through logic optimization. Int. J. Approx. Reason. **45**(3), 488–510 (2007)
11. J.H.R. Jiang, C.C. Lee, A. Mishchenko, C.Y.R. Huang, To SAT or not to SAT: Scalable exploration of functional dependency. IEEE Trans. Comput. **C-59**(4), 457–67 (2010)
12. D.E. Knuth, Fascicle 6: Satisfiability, in *The Art of Computer Programming*, vol. 19 (Addison-Wesley, Reading, 2015)
13. V.N. Kravets, Application of a key-value paradigm to logic factoring. Proc. IEEE **103**(11), 2076–92 (2015)
14. R.R. Lee, J.H.R. Jiang, W.L. Hung, Bi-decomposing large Boolean functions via interpolation and satisfiability solving, in *Proceedings of the 45th Design Automation Conference*, Anaheim, CA (2008), pp. 636–41
15. H.P. Lin, J.H.R. Jiang, R.R. Lee, To SAT or not to SAT: Ashenhurst decomposition in a large scale, in *Proceedings of the International Conference on Computer Aided Design*, San Jose, CA (2008), pp. 32–37
16. E.J. McCluskey, Minimization of Boolean functions. Bell Syst. Tech. J. **35**(6), 1417–44 (1956)
17. K.L. McMillan, Interpolation and SAT-based model checking, in *Proceedings of the International Conference on Computer Aided Verification*, vol. 2725 (Springer, Berlin, 2003), pp. 1–13
18. S. Minato, Fast generation of irredundant sum-of-products forms from binary decision diagrams, in *Proceedings of Synthesis and Simulation Meeting and International Interchange*, Kobe (1992), pp. 64–73
19. A. Mishchenko, R.K. Brayton, SAT-based complete don't-care computation for network optimization, in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich (2005), pp. 412–17
20. A. Mishchenko, R. Brayton, S. Jang, V.N. Kravets, Delay optimization using SOP balancing, in *Proceedings of the International Conference on Computer Aided Design*, San Jose, CA (2011), pp. 375–82
21. A. Morgado, J.P.M. Silva, Good learning and implicit model enumeration, in *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, Hong Kong (2005), pp. 131–36

22. A. Nadel, Generating diverse solutions in SAT, in *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, Ann Arbor, MI (2011), pp. 287–301
23. A. Nadel, V. Ryvchin, Bit-vector optimization, in *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Eindhoven (2016), pp. 851–67
24. A. Petkovska, A. Mishchenko, M. Soeken, G. De Micheli, R. Brayton, P. Ienne, Fast generation of lexicographic satisfiable assignments: Enabling canonicity in SAT-based applications, in *Proceedings of the International Conference on Computer Aided Design*, Austin, TX (2016)
25. J. Rajski, J. Vasudevamurthy, The testability-preserving concurrent decomposition and factorization Boolean expressions. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **11**(6), 778–93 (1992)
26. R.L. Rudell, A.L. Sangiovanni-Vincentelli, Multiple-valued minimization for PLA optimization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **6**(5), 727–50 (1987)
27. S. Sapra, M. Theobald, E.M. Clarke, SAT-based algorithms for logic minimization, in *Proceedings of the 21st IEEE International Conference on Computer Design*, San Jose, CA (2003), p. 510
28. The EPFL Combinational Benchmark Suite, Multi-output PLA benchmarks, http://lsi.epfl.ch/benchmarks
29. G.S. Tseitin, On the complexity of derivation in propositional calculus, in *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970, Symbolic Computation*, ed. by J. Siekmann, G. Wrightson (Springer, Berlin, 1983), pp. 466–83
30. A.K. Verma, P. Brisk, P. Ienne, Iterative layering: Optimizing arithmetic circuits by structuring the information flow, in *Proceedings of the International Conference on Computer Aided Design*, San Jose, CA (2009), pp. 797–804
31. J. Yuan, A. Aziz, C. Pixley, K. Albin, Simplifying Boolean constraint solving for random simulation-vector generation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **23**(3), 412–420 (2004)

Chapter 9
A Branch-and-Bound-Based Minterm Assignment Algorithm for Synthesizing Stochastic Circuit

Xuesong Peng and Weikang Qian

1 Introduction

Stochastic computing (SC) is an alternative to the conventional computing paradigm based on binary radix encoding. In SC, digital circuits are still used to perform computation. However, their inputs are stochastic bit streams [1]. Each stochastic bit stream encodes a value equal to the probability of a 1 in the stream. For example, the stream A shown in Fig. 9.1 encodes the value 0.75.

One major advantage of SC is that it allows complex arithmetic computation to be realized by a very simple circuit. Figure 9.1 shows that arithmetic multiplication can be realized by an AND gate, since for an AND gate, the probability of obtaining a 1 in the output bit stream is equal to the product of the probabilities of obtaining a 1 in the input bit streams.

Since all the bits in the stream have equal weight and a long bit stream is usually used to encode a value, a single bit flip occurring anywhere in the bit stream only causes very small change to the encoded value. Therefore, SC is highly tolerant to bit flip errors [2].

Given its advantages of low hardware cost and strong error tolerance, SC has been used in a number of applications, including image processing [3], decoding of modern error-correcting codes [4], and artificial neural networks [5].

In early days, various elementary computing units in SC were proposed, such as multiplier, scaled adder, divider, and squaring unit [6]. These units were designed manually and can only perform a limited types of computations.

X. Peng • W. Qian (✉)
University of Michigan–Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China
e-mail: sayson@sjtu.edu.cn; qianwk@sjtu.edu.cn

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_9

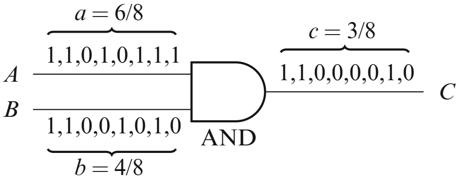


Fig. 9.1 An AND gate performs multiplication on real values encoded by stochastic bit streams

In order to apply SC to a broad range of target computations, several methods to synthesize stochastic circuits have been proposed recently. The works [2, 7, 8] focused on synthesizing reconfigurable stochastic circuits. In [2], the authors proposed a method based on Bernstein polynomial [9] expansion to synthesize combinational logic-based stochastic circuits. In [7] and [8], the authors studied the form of the computation realized by SC using sequential circuits and proposed methods to synthesize such designs. The works [10–12] focused on synthesizing fixed stochastic circuits, which take less area than reconfigurable ones. In [10], the authors demonstrated a fundamental relation between stochastic circuits and spectral transform. Based on this, they proposed a general approach to synthesize stochastic circuits. In [11], the authors found that different Boolean functions could compute the same arithmetic function in SC and proposed the concept of stochastic equivalence class. They proposed a method to search for the optimal Boolean function within an equivalence class. However, their method can only be applied to synthesize multi-linear polynomials. In [12], the authors introduced a general combinational circuit for SC and analyzed its computation. They further proposed a method to synthesize low-cost fixed stochastic circuit to realize a general polynomial.

The study in [12] reveals that in SC, there are a large number of different Boolean functions that realize the same target arithmetic function. Of course, the circuits for different Boolean functions have different costs. In previous work [12], a greedy method was used to find a circuit with low area cost. However, given the extremely large search space, the greedy strategy, although very fast, may not give a minimal solution. In this work, we address this problem by applying a branch-and-bound-based algorithm to extensively search for a Boolean function that will lead to a circuit with low cost. Our approach constructs a function by iteratively adding cubes into the on-set of the Boolean function. The optimal set of cubes to be added is determined through the search process. To improve the runtime, we also introduce a few speed-up techniques.

In summary, the main contributions of our work are as follows.

- We introduce a new method that iteratively selects cubes to form a Boolean function that realizes the target computation in SC.
- We develop a branch-and-bound algorithm to search for the optimal set of cubes to be added.
- We propose several speed-up techniques which prune unpromising branches and significantly improve the runtime of the algorithm.

The rest of the chapter is organized as follows. In Sect. 2, we give the background on the general design proposed in [12] and illustrate the previous synthesis method. We also present the logic synthesis problem for stochastic computing. In Sect. 3, we present the new algorithm. In Sect. 4, we discuss several speed-up techniques. In Sect. 5, we show the experimental results. Finally, we conclude the chapter in Sect. 6.

2 Background on Synthesizing Stochastic Circuits

In this section, we give the background on the general form of the stochastic circuit proposed in [12] and discuss the previous method to synthesize a target function. In what follows, when we say the probability of a signal, we mean the probability of the signal to be a one.

2.1 The General Form and Its Computation

The general form of a stochastic circuit is shown in Fig. 9.2. The circuit is a combinational circuit. It computes an arithmetic function $f(x_1,\ldots,x_n)$, which is encoded by the output bit stream. It has n inputs $X_1, \ldots, X_n$, which are supplied with variable probabilities $x_1, \ldots, x_n$, respectively. In order to offer freedom for realizing different functions, the circuit has m extra inputs $Y_1, \ldots, Y_m$, each supplied with a constant probability of 0.5. They can be easily obtained by a linear feedback shift register (LFSR). The value of m affects the quantization error and is chosen according to the accuracy requirement. The large the value m is, the smaller the quantization error will be.

The study in [12] shows that the general design computes a type of function in the form

$$f(x_1,\ldots,x_n) = \sum_{(a_1,\ldots,a_n)\in\{0,1\}^n} \frac{g(a_1,\ldots,a_n)}{2^m} \prod_{j=1}^{n} x_j^{a_j}\left(1-x_j\right)^{1-a_j}, \qquad (9.1)$$

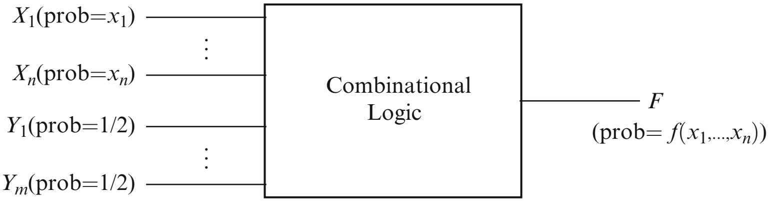


Fig. 9.2 General form of a stochastic circuit [12]

where $0 \leq g(a_1, \ldots, a_n) \leq 2^m$ is an integer. If the combinational circuit realizes a Boolean function $B(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$, then the value $g(a_1, \ldots, a_n)$ is equal to the number of vectors $(b_1, \ldots, b_m) \in \{0, 1\}^m$ such that $B(a_1, \ldots, a_n, b_1, \ldots, b_m)= 1$.

Example 1 Suppose the Boolean function of the combinational circuit in Fig. 9.2 is $B(X_1, X_2, Y_1, Y_2) = X_1Y_1 + X_2Y_2$. Then $B(1, 1, Y_1, Y_2) = Y_1 + Y_2$. Since there are three vectors $(b_1, b_2) \in \{0, 1\}^2$ making $B(1, 1, b_1, b_2) = 1$, the value $g(1, 1) = 3$. Similarly, we can derive $g(0, 0) = 0$, $g(0, 1) = 2$, and $g(1, 0) = 2$. Since $m = 2$, according to Eq. (9.1), the output function is

$$f(x_1, x_2) = \frac{1}{2}(1 - x_1)x_2 + \frac{1}{2}x_1(1 - x_2) + \frac{3}{4}x_1x_2. \tag{9.2}$$

□

The function of the form shown in Eq. (9.1) is called a *binary combination polynomial* (BCP) [12]. If we expand a BCP, we can obtain a *multi-linear polynomial* (MLP) of the following form

$$f(x_1, \ldots, x_n) = \sum_{(a_1,\ldots,a_n)\in\{0,1\}^n} \frac{c(a_1, \ldots, a_n)}{2^m} \prod_{j=1}^{n} x_j^{a_j}, \tag{9.3}$$

where $c(a_1, \ldots, a_n)$'s are integers. The degree of each variable in an MLP is at most 1. For example, expanding Eq. (9.2), we can obtain an MLP

$$f(x_1, x_2) = \frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{1}{4}x_1x_2. \tag{9.4}$$

2.2 Synthesis of General Function

Given a target function, a procedure was proposed in [12] to synthesize a stochastic circuit of the general form to realize that function. We use an example to illustrate the procedure. Since the computation realized by a general-form stochastic circuit is a polynomial, the target function will be first approximated as a polynomial.

Now suppose the polynomial is $f = \frac{1}{4}x_1^2 + \frac{1}{2}x_2$. Next, it will be transformed into an MLP. This is achieved by introducing two new variables $x_{1,1}$ and $x_{1,2}$ with their values both set as x_1. The MLP obtained is

$$f = \frac{1}{4}x_{1,1}x_{1,2} + \frac{1}{2}x_2. \tag{9.5}$$

The next step is to map the MLP into a BCP. By a procedure shown in [12], the result is

$$\begin{aligned} f = &\frac{1}{2}(1-x_{1,1})(1-x_{1,2})x_2 + \frac{1}{2}(1-x_{1,1})x_{1,2}x_2 \\ &+ \frac{1}{2}x_{1,1}(1-x_{1,2})x_2 + \frac{1}{4}x_{1,1}x_{1,2}(1-x_2) + \frac{3}{4}x_{1,1}x_{1,2}x_2. \end{aligned} \tag{9.6}$$

Assume that the number of Y-variables is $m = 2$ and the Boolean function is $B(X_{1,1}, X_{1,2}, X_2, Y_1, Y_2)$. Comparing Eq. (9.6) with Eq. (9.1), we can obtain that the Boolean function should satisfy that

$$\begin{aligned} &g(0,0,0) = 0,\ g(0,0,1) = 2,\ g(0,1,0) = 0,\ g(0,1,1) = 2, \\ &g(1,0,0) = 0,\ g(1,0,1) = 2,\ g(1,1,0) = 1,\ g(1,1,1) = 3. \end{aligned} \tag{9.7}$$

However, since $x_{1,1} = x_{1,2} = x_1$, the terms $(1-x_{1,1})x_{1,2}x_2$ and $x_{1,1}(1-x_{1,2})x_2$ are the same. Also, the terms $(1-x_{1,1})x_{1,2}(1-x_2)$ and $x_{1,1}(1-x_{1,2})(1-x_2)$ are the same. Therefore, the requirement for the Boolean function can be relaxed as follows:

$$\begin{aligned} &g(0,0,0) = 0,\ g(0,0,1) = 2,\ g(0,1,0) + g(1,0,0) = 0, \\ &g(0,1,1) + g(1,0,1) = 4,\ g(1,1,0) = 1,\ g(1,1,1) = 3. \end{aligned} \tag{9.8}$$

In the general case, suppose the target polynomial has k variables $x_1, \ldots, x_k$ and the degree of x_i is d_i, for $i = 1, \ldots, k$. Define $n = \sum_{i=1}^{k} d_i$. To transform the original target into an MLP, we will introduce n new variables $x_{1,1}, \ldots, x_{1,d_1}, \ldots, x_{i,1}, \ldots, x_{i,d_i}, \ldots, x_{k,1}, \ldots, x_{k,d_k}$, with the values of $x_{i,1}, \ldots, x_{i,d_i}$ all set to x_i. The BCP has 2^n product terms of the form

$$\prod_{i=1}^{k}\prod_{j=1}^{d_i} x_{i,j}^{a_{i,j}}(1-x_{i,j})^{1-a_{i,j}}, \tag{9.9}$$

where $(a_{1,1}, \ldots, a_{1,d_1}, \ldots, a_{k,1}, \ldots, a_{k,d_k}) \in \{0,1\}^n$. Each product term has a one-to-one correspondence to a vector $(a_{1,1}, \ldots, a_{1,d_1}, \ldots, a_{k,1}, \ldots, a_{k,d_k}) \in \{0,1\}^n$. We call the vector *the characteristic vector* of the product term. We partition the set $\{0,1\}^n$ into $\prod_{i=1}^{k}(1+d_i)$ *equivalence classes* $I(s_1, \ldots, s_k)$, $0 \le s_1 \le d_1, \ldots, 0 \le s_k \le d_k$, where

$$I(s_1, \ldots, s_k) = \left\{ (a_{1,1}, \ldots, a_{k,d_k}) \in \{0,1\}^n : \sum_{j=1}^{d_i} a_{i,j} = s_i, \text{for all } i = 1, \ldots, k \right\}. \tag{9.10}$$

Under the condition that for all $1 \le i \le k$, $x_{i,1} = \cdots = x_{i,d_i} = x_i$, two product terms are the same if and only if their characteristic vectors belong to the same equivalence class. Therefore, to realize the target polynomial, we only require that the sum of the g values over all the vectors in an equivalence class is equal to a specific constant. Mathematically, the requirement is that for all $0 \le s_1 \le d_1, \ldots,$ $0 \le s_k \le d_k$

$$\sum_{(a_{1,1},\ldots,a_{k,d_k}) \in I(s_1,\ldots,s_k)} g(a_{1,1}, \ldots, a_{k,d_k}) = G(s_1, \ldots, s_k), \tag{9.11}$$

where $0 \le G(s_1, \ldots, s_k) \le 2^m \prod_{i=1}^{k} \binom{d_i}{s_i}$ is a constant that can be derived by adding up the corresponding g values of an initial BCP transformed from the original target function.

The example shown before corresponds to a situation in which $k = 2$, $d_1 = 2$, and $d_2 = 1$. Then we have six equivalence classes

$$\begin{gathered} I(0,0) = \{(0,0,0)\}, \quad I(0,1) = \{(0,0,1)\}, \quad I(1,0) = \{(0,1,0),(1,0,0)\}, \\ I(1,1) = \{(0,1,1),(1,0,1)\}, \quad I(2,0) = \{(1,1,0)\}, \quad I(2,1) = \{(1,1,1)\}. \end{gathered} \tag{9.12}$$

Given the above equivalence classes, the requirement on the g values specified by Eq. (9.11) is same as Eq. (9.8) we derived before.

2.3 The Circuit Synthesis Problem

Equation (9.11) shows a requirement on the Boolean function to realize the target polynomial. However, there are a large number of Boolean functions that can satisfy the requirement. In order to synthesize an optimal circuit, we need to find an optimal Boolean function that satisfies the requirement. For simplicity, we focus on two-level circuit in this work and we use the literal number of the sum-of-product (SOP) form as the cost measure. The optimization problem is stated as follows.

> Given an integer m and $\prod_{i=1}^{k}(1 + d_i)$ integers $G(0, \ldots, 0), \ldots, G(d_1, \ldots, d_k)$ such that $0 \le G(s_1, \ldots, s_k) \le 2^m \prod_{i=1}^{k} \binom{d_i}{s_i}$ for any $0 \le s_1 \le d_1, \ldots,$ $0 \le s_k \le d_k$, determine an optimal Boolean function such that its g values satisfy Eq. (9.11).

Fig. 9.3 The matrix representation of the Boolean function $B(X_1, X_2, X_3, Y_1, Y_2) = \overline{X_1}\,\overline{Y_1} + X_2\overline{Y_1} + \overline{X_1}X_3$

$Y \backslash X$	000	001	011	010	110	111	101	100
00	1	1	1	1	1	1		
01	1	1	1	1	1	1		
11		1	1					
10		1	1					

The above problem has flexibility in determining the final Boolean function. However, it is different from the traditional logic minimization with don't cares or Boolean relation minimization problem [13]. The problem we consider here has a constraint on the number of input vectors belonging to a subset that make the function evaluate to 1. Thus, the determination of the output for an input vector will reduce the output choices of the other input vectors belonging to the same subset. In contrast, logic minimization with don't cares or Boolean relation minimization does not have that constraint. The determination of the output of an input vector does not reduce the output choices for the other input vectors. Therefore, solving the above problem requires a new method.

Suppose the Boolean function is $B(X_{1,1}, \ldots, X_{1,d_1}, \ldots, X_{k,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m)$. We represent it using a matrix, where the columns represent the X-variables and the rows represent the Y-variables. Both the columns and the rows are arranged in Gray code order. An example is shown in Fig. 9.3 for a case where $k = 1$, $d_1 = 3$, and $m = 2$.

Using that matrix representation, the number $g(a_{1,1}, \ldots, a_{k,d_k})$ is equal to the number of ones in the column $a_{1,1} \ldots a_{k,d_k}$. Then the optimization problem is to distribute $G(s_1, \ldots, s_n)$ ones to columns corresponding to the vectors in the class $I(s_1, \ldots, s_n)$ to achieve an optimal Boolean function. A method was proposed in the previous work [12] to find a good solution. It applies a greedy strategy to distribute the ones. Assume $l = \lfloor G(s_1, \ldots, s_n)/2^m \rfloor$. Then the method sets the g values of the first l vectors in the class $I(s_1, \ldots, s_n)$ as 2^m, the g value of the $(l + 1)$-th vector as $(G(s_1, \ldots, s_n) - 2^m l)$, and the g values of the remaining vectors as 0. The following example illustrates how the previous method works.

Example 2 Consider a case where $k = 1$, $d_1 = 3$, and $m = 2$. There are four equivalence classes for this case:

$$\begin{aligned} &I(0) = \{(0,0,0)\}, \quad I(1) = \{(0,0,1), (0,1,0), (1,0,0)\}, \\ &I(2) = \{(0,1,1), (1,0,1), (1,1,0)\}, \quad I(3) = \{(1,1,1)\}. \end{aligned} \tag{9.13}$$

Assume the sums of g values over all the vectors in each equivalence class are $G(0) = 2$, $G(1) = 6$, $G(2) = 6$, and $G(3) = 2$. For equivalence classes $I(0)$ and $I(3)$, each of them covers one column. We set $g(0,0,0) = 2$ and $g(1,1,1) = 2$.

Fig. 9.4 The matrix representation of the Boolean function $B(X_1, X_2, X_3, Y_1, Y_2) = \overline{Y_1}$

$Y\backslash X$	000	001	011	010	110	111	101	100
00	1	1	1	1	1	1	1	1
01	1	1	1	1	1	1	1	1
11								
10								

For equivalence classes $I(1)$ and $I(2)$, each of them covers three columns. Since $\lfloor G(1)/2^m \rfloor = 1$, we assign $g(0,0,1) = 4$, $g(0,1,0) = 2$, and $g(1,0,0) = 0$. Similarly, for class $I(2)$, we assign $g(0,1,1) = 4$, $g(1,1,0) = 2$, and $g(1,0,1) = 0$. The final assignment of the ones is shown in Fig. 9.3. The Boolean function is $B = \overline{X_1}\,\overline{Y_1} + X_2\overline{Y_1} + \overline{X_1}X_3$, which has six literals. □

However, the previous method may not give an optimal solution. For the case shown in Example 2, a better assignment is shown in Fig. 9.4, which gives a function $B = \overline{Y_1}$. In this work, we explore a better solution to the optimization problem.

3 The Proposed Algorithm

In this section, we present the new algorithm. For simplicity, we focus on univariate polynomials, i.e., $k = 1$. Our work can be extended to handle multivariate polynomials. The only difference is that there are more equivalence classes for multivariate cases. For univariate case, we have $n = d_1$ and we assume the n X inputs are $X_1, X_2, \ldots, X_n$.

The basic approach we use to construct an optimal solution is to add cubes one by one into the on-set of the Boolean function. Although the previous work also uses this strategy, it only adds cubes which cover minterms in the same equivalence class. In contrast, our method also adds cubes across different equivalence classes.

3.1 Preliminaries

Before presenting the details, we first introduce a few notations and definitions. We use $M(a_1, \ldots, a_n, b_1, \ldots, b_m)$ to denote the minterm corresponding to an input vector $(a_1, \ldots, a_n, b_1, \ldots, b_m) \in \{0,1\}^{n+m}$. We say a minterm $M(a_1, \ldots, a_n, b_1, \ldots, b_m)$ is in an equivalence class $I(i)$ $(0 \le i \le n)$ if $(a_1, \ldots, a_n) \in I(i)$.

We use a vector $(v_0, \ldots, v_n)$ to represent numbers of unassigned minterms for $(n+1)$ equivalent classes. We call such a vector *problem vector*. Initially, the problem vector is equal to $(G(0), \ldots, G(n))$, given by the problem specification. With cubes added into the on-set, the entries in the problem vector will be reduced. Eventually, when all the minterms have been decided, the problem vector will become a zero vector.

Fig. 9.5 Two different cubes of the same cube vector $[0, 2, 2]$. (**a**) Cube X_1. (**b**) Cube X_2

$Y_1\backslash X_1X_2$	00	01	11	10
0			1	1
1			1	1

(a)

$Y_1\backslash X_1X_2$	00	01	11	10
0		1	1	
1		1	1	

(b)

We can also represent a cube by a vector of length $(n + 1)$. It is formed by the numbers of minterms of the cube in each equivalence class. We call such a vector *cube vector*. In order to distinguish it from the problem vector, we represent the cube vector using square brackets. For example, assume that $n = 2$ and $m = 1$. The cube X_1 contains four minterms $X_1X_2\overline{Y_1}$, $X_1\overline{X_2}\,\overline{Y_1}$, $X_1X_2Y_1$, and $X_1\overline{X_2}Y_1$, as shown in Fig. 9.5a. The minterms $X_1\overline{X_2}\,\overline{Y_1}$ and $X_1\overline{X_2}Y_1$ are in the equivalence class $I(1)$ and the minterms $X_1X_2\overline{Y_1}$ and $X_1X_2Y_1$ are in the equivalence class $I(2)$. There are no minterms of the cube X_1 in the equivalence class $I(0)$. Therefore, the vector of the cube X_1 is $[0, 2, 2]$. Note that although each cube has a unique cube vector, a cube vector may correspond to a number of different cubes. For example, the cube X_2 has the same cube vector as the cube X_1, as shown in Fig. 9.5b.

Our approach splits the problem vector into a set of cube vectors. In order to manipulate on the vector, it is important to know the valid form of a cube vector. We have the following claim on this.

Theorem 1 *A cube vector is of the form* $[0, \ldots, 0, 2^l\binom{r}{0}, 2^l\binom{r}{1}, \ldots, 2^l\binom{r}{r}, 0, \ldots, 0]$*, where* $0 \le r \le n$ *and* $0 \le l \le m$ *are the numbers of the missing X-variables and missing Y-variables in the cube, respectively. The cube vector has t zeros at the beginning and* $(n - t - r)$ *zeros at the end, where* $0 \le t \le n - r$ *is equal to the number of uncomplemented X-variables in the cube and* $(n - t - r)$ *is equal to the number of complemented X-variables in the cube.*

Proof Consider the matrix representation of the cube. Since there are l missing Y-variables in the cube, the cube covers 2^l rows and all the covered rows have the same pattern. Note that each covered row is also a cube, which contains all the m Y-variables. Therefore, we only need to show that for such a cube, its cube vector is of the form $[0, \ldots, 0, 2^l\binom{r}{0}, 2^l\binom{r}{1}, \ldots, 2^l\binom{r}{r}, 0, \ldots, 0]$.

We consider the X-variables of the cube. Suppose that there are t uncomplemented X-variables and r missing X-variables in the cube. Then, the cube has $(n - t - r)$ complemented X-variables. The cube covers 2^r minterms, among which $\binom{r}{i}$ minterms are in the equivalence class $I(t + i)$, for $i = 0, \ldots, r$. For any $0 \le j < t$ or $t + r < j \le n$, there are no minterms of the cube in the equivalence class $I(j)$. Therefore, the cube vector is of the form $[0, \ldots, 0, \binom{r}{0}, \binom{r}{1}, \ldots, \binom{r}{r}, 0, \ldots, 0]$, in which there are t zeros at the beginning and $(n - t - r)$ zeros at the end. □

Example 3 Assume that $n = 3$ and $m = 2$. Then, the cube X_1Y_1 contains 8 minterms $X_1\overline{X_2}\,\overline{X_3}Y_1\overline{Y_2}$, $X_1\overline{X_2}\,\overline{X_3}Y_1Y_2$, $X_1\overline{X_2}X_3Y_1\overline{Y_2}$, $X_1\overline{X_2}X_3Y_1Y_2$, $X_1X_2\overline{X_3}Y_1\overline{Y_2}$, $X_1X_2\overline{X_3}Y_1Y_2$, and $X_1X_2X_3Y_1\overline{Y_2}$, $X_1X_2X_3Y_1Y_2$. Its cube vector is $[0, 2, 4, 2] = \left[0, 2\binom{2}{0}, 2\binom{2}{1}, 2\binom{2}{2}\right]$. For this cube vector, $l = 1$ is equal to the number of missing Y-variables and $r = 2$ is equal to the number of missing X-variables. The number of zeros at the beginning is 1, which is equal to the number of uncomplemented X-variables in the cube. The number of zeros at the end is 0, which is equal to the number of complemented X-variables in the cube. □

3.2 *The Basic Idea*

As mentioned at the beginning of this section, our approach iteratively adds cubes into the on-set of the Boolean function. Each time a cube is added, some entries in the problem vector will be reduced. When the problem vector becomes zero, the Boolean function is constructed.

Generally, a cube added later may intersect with a cube added previously. However, in our approach, we restrict that a cube added later should be disjoint to any cubes added before. For simplicity, we call this restriction *disjointness constraint*. Although this restriction may cause some quality loss, it has two benefits. First, it makes the counting of minterms easy, because we do not need to consider the overlapped minterms. With a cube satisfying the disjointness constraint added, the problem vector can be easily updated by subtracting the cube vector from the original problem vector. Second, the constraint eliminates many redundant cases. For example, adding two non-disjoint cubes X_1 and X_2 is equivalent to adding two disjoint cubes X_1 and $\overline{X_1}X_2$. Note that although the Boolean function is constructed by adding disjoint cubes, the final Boolean function will be further simplified by the two-level logic optimization tool ESPRESSO [14]. Thus, the final result is a set of non-disjoint cubes corresponding to a minimum SOP expression.

In each iteration, when picking a cube, we also require that each entry in the cube vector of the cube is no larger than the corresponding entry in the current problem vector. For simplicity, we call this constraint *capacity constraint*. If a cube satisfies both the disjointness constraint and the capacity constraint, we say the cube is *valid*.

In each iteration, we apply a greedy strategy in choosing the cube to be added: we choose the largest cube among all valid cubes. The reasons for this are (1) in two-level logic synthesis, larger cubes have fewer literals and (2) with the largest cubes added, the problem vector is reduced most. The details of how we choose the largest valid cube will be discussed in Sect. 3.3. The procedure of choosing the largest valid cube involves obtaining a cube corresponding to the cube vector, which will be discussed in Sect. 3.4. Since at each iteration, there may exist more than one largest valid cube for the current problem setup, we actually apply a branch-and-bound algorithm to find the optimal solution, which will be discussed in Sect. 3.5.

3.3 *Selecting the Largest Valid Cube*

Suppose that at the beginning of one iteration, the problem vector is $(v_0, \dots, v_n)$. Let s be the sum of all the entries in the problem vector, i.e., $s = \sum_{i=0}^{n} v_i$. Assume

$q = \lfloor \log_2 s \rfloor$. Since the largest valid cube satisfies the capacity constraint, it contains at most 2^q minterms. Our method to find the largest valid cube first checks whether there exists a valid cube with 2^q minterm.

According to Theorem 1, the cube vector should be in the form of $[0, \ldots, 0, 2^l\binom{r}{0}, 2^l\binom{r}{1}, \ldots, 2^l\binom{r}{r}, 0, \ldots, 0]$, where $0 \le r \le n$ and $0 \le l \le m$. Furthermore, since the cube contains 2^q minterms, we require that $l + r = q$. We will examine all cube vectors that satisfy the above two requirements and keep those which also satisfy the capacity constraint. Then, for each kept cube vector, we will check whether it has a corresponding cube that satisfies the disjointness constraint. The details of how to check the existence of such a cube will be discussed in Sect. 3.4. If such a cube exists, it is a largest valid cube.

Example 4 Suppose $n = 2$, $m = 2$, and we are given an initial problem vector of $(2, 5, 2)$. The sum of all the entries in the problem vector is nine. Thus, the largest valid cube has at most eight minterms. We first check whether there exists any valid cube with 8 minterms. This type of cubes should be in the form of $[0, \ldots, 0, 2^l\binom{r}{0}, 2^l\binom{r}{1}, \ldots, 2^l\binom{r}{r}, 0, \ldots, 0]$ with $0 \le r \le 2$, $0 \le l \le 2$, and $l + r = 3$. Given the constraint, we have either $l = 2$ and $r = 1$ or $l = 1$ and $r = 2$. Thus, the possible cube vectors are $[0, 4, 4]$, $[4, 4, 0]$, and $[2, 4, 2]$. Among these three cube vectors, only the cube vector $[2, 4, 2]$ satisfies the capacity constraint. Then, we will further check whether it has a corresponding cube satisfying the disjointness constraint. Since no cubes have been added yet, we can find a valid cube for the cube vector $[2, 4, 2]$, for example, the cube Y_1. This cube is one largest valid cube. □

In some situations, there may not exist a valid cube with 2^q minterms because either the capacity constraint or the disjointness constraint is violated. The following is an example.

Example 5 Suppose $n = 2$, $m = 3$, and we are given an initial problem vector of $(1, 3, 7)$. The sum of all the entries in the problem vector is 11. Thus, the largest valid cube has at most eight minterms. The possible cube vectors of eight minterms are $[0, 0, 8]$, $[0, 8, 0]$, $[8, 0, 0]$, $[0, 4, 4]$, $[4, 4, 0]$, and $[2, 4, 2]$. However, none of these cube vectors satisfy the capacity constraint. Therefore, we cannot find a valid cube with eight minterms. □

If there exists no valid cube with 2^q minterms, then we will reduce the minterm number by half and check whether there exists a valid cube with 2^{q-1} minterms. This procedure will be repeated until we are able to find a valid cube with 2^i minterms for some $0 \le i \le q$. Then, that cube is the largest valid cube. Since in the worst case, we can always find a minterm that is valid, the procedure guarantees to terminate at some point.

However, in general cases, the largest valid cube is not unique. This is due to the existence of more than one largest cube vector that satisfies the capacity constraint and the existence of more than one cube for a cube vector.

Example 6 Suppose $n = 2$, $m = 3$, and we are given an initial problem vector of $(4, 8, 3)$. The largest possible cube has eight minterms. Among all cube vectors of eight minterms, three satisfy the capacity constraint: $[0, 8, 0]$, $[4, 4, 0]$, and

$[2, 4, 2]$. Furthermore, there exists more than one cube that satisfies the disjointness constraint for each of the three cube vectors. For example, for the cube vector $[0, 8, 0]$, it corresponds to cubes $X_1\overline{X_2}$ and $\overline{X_1}X_2$, which satisfy the disjoint constraint. Therefore, there exist more than one largest valid cubes for this case. □

When there are multiple choices of the largest valid cubes, we want to evaluate all of them and choose the best one. For this purpose, we apply a branch-and-bound algorithm to find an optimal Boolean function. The details of it will be discussed in Sect. 3.5.

3.4 Obtaining Cubes for a Cube Vector

In this section, we discuss one important procedure in selecting the largest valid cube: obtaining cubes for a given cube vector that satisfies the disjointness constraint. Since a cube is composed of X-variables and Y-variables, the procedure is divided into two parts: determining the X-variables and determining the Y-variables.

The X-variables are determined based on the form of the cube vector. As shown in Theorem 1, if the vector is of the form $[0, \ldots, 0, 2^l\binom{r}{0}, 2^l\binom{r}{1}, \ldots, 2^l\binom{r}{r}, 0, \ldots, 0]$ where there are t zeros at the beginning and $(n - t - r)$ zeros at the end, then the set of X-variables is composed of t uncomplemented X-variables and $(n - t - r)$ complemented X-variables. For example, if $n = 3$ and the cube vector is of the form $[0, 4, 4, 0]$, then the possible X-variable cubes are $X_1\overline{X_2}$, $X_1\overline{X_3}$, $X_2\overline{X_1}$, $X_2\overline{X_3}$, $X_3\overline{X_1}$, and $X_3\overline{X_2}$.

Next, for each set of possible X-variables, we will further determine all sets of Y-variables so that the cube formed by these X-variables and Y-variables satisfies the disjointness constraint. According to Theorem 1, the set of Y-variables we need to pick consists of $(m - l)$ Y-variables. To obtain all valid sets of Y-variables, we can simply enumerate all cubes consisting of $(m - l)$ Y-variables and keep those when combined with the X-variable cube do not overlap with the current Boolean function. However, we could find a large number of valid Y-variable cubes, which increases the number of largest valid cubes. In order to reduce the choices, in our implementation, we enumerate all cubes with $(m - l)$ Y-variables in the Gray code order and keep the first valid Y-variable cube for each set of possible X-variables.

3.5 Branch-and-Bound Algorithm

As we mentioned before, in each iteration, there may exist more than one largest valid cube. If this happens, it is hard to decide which one will minimize the literal number of the final Boolean function. Therefore, we apply a branch-and-bound algorithm to evaluate all possible cube choices. An example of the search tree is shown in Fig. 9.6. Each leaf of the search tree corresponds to a final solution, represented by a set of cubes. Each internal node stores a partial solution composed

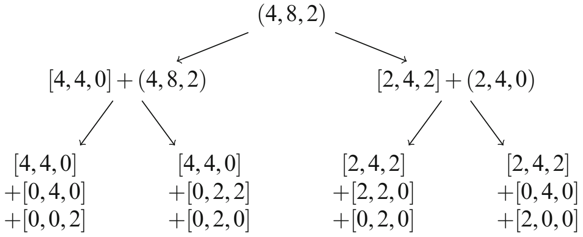


Fig. 9.6 An illustration of the solution tree for the problem with the initial problem vector $(4, 8, 2)$ and $m = 2$. Note that for simplicity, we use a cube vector to represent a cube and we only show a partial tree

Algorithm 1 Branch-and-bound algorithm to find optimal function

```
Input: problem vector $v = (G_0, \ldots, G_n)$ and an integer $m$
Output: the set of cubes of the final Boolean function $B$
initialize a node $N$: $N.vector \leftarrow v$; $N.cubeset \leftarrow \emptyset$;
initialize the optimal literal number $n_o \leftarrow \infty$;
initialize the optimal cube set $S_o \leftarrow \emptyset$;
push the node $N$ into an empty stack $Stk$;
while $Stk$ is not empty do
    pop a node $N$ out of $Stk$;
    find a list $L$ of largest valid cubes for $N.vector$, $N.cubeset$, and $m$;
    for each cube $C$ in $L$ do
        if $litcount(N.cubeset \cup C) < n_o$ then
            $N_{new}.vector \leftarrow N.vector - vector(C)$;
            $N_{new}.cubeset \leftarrow N.cubeset \cup C$;
            if $N_{new}.vector = 0$ then // reach a leaf
                $n_o \leftarrow litcount(N_{new}.cubeset)$;
                $S_o \leftarrow N_{new}.cubeset$;
            else
                push the node $N_{new}$ into $Stk$;
            end if
        end if
    end for
end while
return $S_o$;
```

of a set of cubes added and the remaining problem vector. The root is the initial problem vector. At each internal node, the multiple choices of the largest valid cubes for the current problem vector lead to multiple branches from the node.

In order to apply a brand-and-bound algorithm, we need a lower bound on the candidate solutions from a branch. We choose the lower bound as the minimum literal number for the set of cubes that forms a partial solution at a branch. For example, for the branch $[2, 4, 2] + (2, 4, 0)$ shown in Fig. 9.6, its lower bound is the minimum literal number for the cube with the cube vector $[2, 4, 2]$. Strictly speaking, the minimum literal number for the set of chosen cubes at a branch may not be the lower bound for that branch, because with more cubes determined later, it is possible to reduce the literal count due to cube expansion and redundant cube removal. However, since the cubes selected later are no larger than any of the cubes already chosen, it is more likely that with more cubes selected, the literal count will increase. Thus, we use the proposed method to obtain the lower bound. A branch will be pruned if the lower bound for the branch is larger than or equal to the minimum literal count for the best solution obtained so far. In practice, the exact minimum literal number for a set of cubes is computationally expensive to obtain. Instead, we call the powerful two-level logic optimization tool ESPRESSO [14] to estimate the minimum value. Algorithm 1 summarizes the proposed branch-and-bound algorithm to find an optimal solution. Note that we explore the solution tree using the depth-first traversal.

4 Speed-Up Techniques

Although the branch-and-bound algorithm deletes some unpromising branches, there are still too many branches to process as the degree of the polynomial increases, which increases the runtime considerably. However, there are numerous branches unnecessary to process, either because they are unpromising or because they produce the same results. In this section, we present several techniques to speed up the algorithm with only small quality loss.

4.1 Removing Branches with Duplicated Cube Sets

For a node in the search tree, even though the sum of all entries in its problem vector is in the interval $[2^q, 2^{q+1} - 1]$, the size of the largest valid cube may not be 2^q. Example 5 shows such a case. If this happens, we may add in sequence multiple cubes of the same size of 2^u, where $u < q$ is an integer. In the original branch-and-bound algorithm, the order that these cubes are added can produce different branches. Nevertheless, in most cases, different orders will finally lead to the same results.

Example 7 Suppose $n = 2$, $m = 3$, and the initial problem vector is $(1, 6, 2)$. We cannot extract a valid cube of size 8 from the initial problem vector. As a result, the largest valid cube is of size 4. Its cube vector is either $[1, 2, 1]$ or $[0, 4, 0]$. With the original algorithm, if the first cube selected is of the cube vector $[1, 2, 1]$, then the second cube selected will be of the cube vector $[0, 4, 0]$. On the other hand, if the first cube selected is of the cube vector $[0, 4, 0]$, then the second cube selected will be of the cube vector $[1, 2, 1]$. These two branches from the root node will produce the same results. □

Those branches with the same set of cubes as a branch explored before are unnecessary to be explored again. To remove them, we keep track of the sets of cube vectors we have already examined. If the set of the cube vectors at the current branch has been examined before, the branch will be pruned.

4.2 Bounding by the Optimal Cost at Each Level

In the original algorithm, a branch is pruned only when its lower bound exceeds the value of the optimal full solution known so far. In practice, given that each time we always add a largest valid cube, it is very likely that for any level i in the search tree, the cost of the partial solution at level i in a branch that will be pruned later is larger than the cost of the optimal partial solution at level i. In other words, only those branches with costs close to the optimal partial solution at each level are promising in leading to the optimal full solution. Therefore, we propose another speed-up technique which prunes branches based on the cost of the optimal partial solution at each level. With this technique, we can find and prune many unpromising branches earlier. However, the proposed method is just a heuristic. In order to reduce the quality loss caused by applying this heuristic, we choose the bound at each level as the cost of the optimal partial solution at the current level multiplied by a constant $m_l > 1$. We will only delete those branches whose costs exceed the bound. In real implementation, since we traverse the solution tree in a depth-first way, the optimal partial solution is obtained among all the explored nodes at the current level.

4.3 Limiting Update Count and Explored Node Number

The previous two speed-up techniques focus on eliminating unpromising branches. However, for some extreme cases, the numbers of nodes explored could still be very large. In order to further reduce the runtime for these extreme cases, we impose limits on the update count and the number of explored nodes.

Our algorithm will update the optimal solution if the current solution is no worse than the optimal one recorded. As a result, each update will either improve the result or leave it unchanged. Our experimental results showed that with more updates, the improvement will gradually reduce. Therefore, we consider the solution to be optimal enough after a specific number of updates. Thus, we set a limit on the update number and terminate the algorithm once the limit is reached. From our experimental results, we set this limit as three. The quality loss is negligible.

Even though limiting the updating number can further improve the runtime for some extreme cases, there are still some cases for which a large number of nodes are explored between two consecutive updates. In our experiment, there is a recorded case for which after the second update, the algorithm processed 16,463 other nodes to reach the third update. It took about 57 min to explore these nodes, but no improvement was made for the third update. Therefore, we also set a limit on the number of explored nodes. The algorithm records the number of nodes explored. Once the initial solution has been found, the number of nodes explored will be compared against the limit and the algorithm will terminate once the limit is reached. In our experiment, the limit is often set from 15 to 30 for $3 \leq n \leq 7$ and $3 \leq m \leq 7$, or larger if needed. With a larger limit, we can achieve a better solution.

5 Experiment Results

In this section, we show the experimental results of the proposed algorithm. All the experiments were conducted on a desktop with 3.20 GHz Intel® Core™ i5-4570 CPU and 16.0 GB RAM. ESPRESSO is used to evaluate the literal count [14].

We applied the proposed branch-and-bound algorithm with the speed-up technique to univariate polynomials with $3 \leq n \leq 7$ and $3 \leq m \leq 7$. For each pair of n and m, we generated 50 random cases and obtained the average result. Table 9.1 shows for each pair of n and m, the average percentage of literal count reduction

Table 9.1 The average percentage of literal count reduction by the proposed algorithm over the previous method [12] (in the first row of each cell), the percentage of improved and unchanged cases (in the second row of each cell), and the average runtime of the proposed algorithm (in the third row of each cell) for different pairs of n and m

$m \backslash n$	3	4	5	6	7
3	6%	12%	19%	18%	26%
	100%	90%	94%	88%	100%
	0.44 s	0.60 s	1.60 s	2.56 s	5.20 s
4	3%	13%	16%	22%	29%
	98%	100%	88%	92%	94%
	0.60 s	1.42 s	2.46 s	4.20 s	7.48 s
5	2%	13%	18%	18%	26%
	92%	100%	92%	84%	88%
	0.88 s	1.14 s	2.92 s	8.74 s	17.1 s
6	2%	12%	15%	18%	23%
	92%	96%	86%	88%	90%
	1.34 s	3.90 s	7.04 s	16.6 s	42.6 s
7	2%	10%	14%	17%	22%
	88%	94%	86%	90%	90%
	2.46 s	8.54 s	16.6 s	39.2 s	116 s

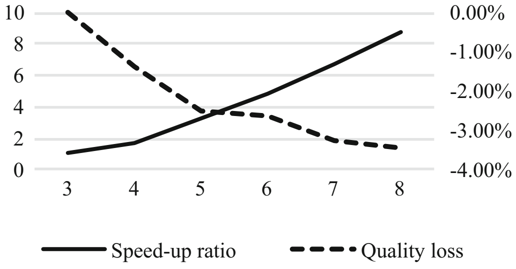


Fig. 9.7 Comparison between the branch-and-bound algorithm without acceleration and the accelerated algorithm for $n = 3$ and $3 \leq m \leq 8$

by the proposed algorithm over the method in [12], the percentage of improved or unchanged cases among all 50 cases, and the average runtime in seconds of the proposed algorithm. The literal reduction percentage, the percentage of improved and unchanged cases, and the runtime are shown in the first row, the second row, and the third row of each cell, respectively. For example, for $n = 4$ and $m = 4$, the proposed algorithm saves 13% literal count on average. 100% of the 50 cases have their literal counts reduced or unchanged. The average runtime is 1.42 s.

It can be seen that in the average sense, the proposed algorithm reduces the literal count compared to the previous method. When n is small, the literal count reduction is small because the previous greedy method is able to find a good solution among limited choices. However, as n increases, more percentage of literals is saved. For $n = 7$, the literal saving reaches up to 29%. For each pair of n and m, at least 84% of cases have their literal counts improved or unchanged. For some pairs of n and m, all 50 cases have their literal counts improved or unchanged. With the increase of n and m, the runtime also increases, which is due to the growth of the search space. Notice that the runtime of the previous method is negligible compared to ours, due to its greedy nature. However, since the values n and m for a typical stochastic circuit tend to be small, the runtime of our algorithm is still affordable for a normal stochastic circuit. In summary, in situations where better circuit quality is pursued, our method gives a better solution under a reasonable amount of runtime.

We also compared the proposed accelerated algorithm to the branch-and-bound algorithm without using the speed-up techniques. Due to the inefficiency of the algorithm without acceleration, the comparison was only done for polynomials of degree $n = 3$ and $3 \leq m \leq 8$. Figure 9.7 plots the speed-up ratio (shown in solid line, y-axis on the left) and the quality loss (shown in dashed line, y-axis on the right) of the accelerated algorithm for different m values. For the quality loss, the more negative the value is, the more loss the accelerated algorithm has. We can see from Fig. 9.7 that as the problem instance grows, more runtime can be saved through the speed-up techniques. However, the quality loss also increases. Nevertheless, the quality loss is small. Indeed, in terms of the absolute value, the average quality loss is smaller than one literal. Thus, the speed-up techniques have a negligible impact on the quality.

6 Conclusion

In this work, we proposed a search-based method for synthesizing stochastic circuits. The synthesis problem we considered here is different from the traditional logic synthesis problem in that there exist many different Boolean functions to realize a target computation. We proposed a branch-and-bound algorithm to systematically explore the solution space. A final solution is obtained by adding a series of cubes to the on-set of the Boolean function. We also provided several speed-up techniques. The experimental results showed that our algorithm produced smaller circuits than a previous greedy approach, especially when the target polynomial had a high degree.

Acknowledgements This work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61204042 and 61472243.

References

1. B.R. Gaines, Stochastic computing systems, in *Advances in Information Systems Science* (Springer, New York, 1969), pp. 37–172
2. W. Qian, X. Li, M.D. Riedel, K. Bazargan, D.J. Lilja, An architecture for fault-tolerant computation with stochastic logic. IEEE Trans. Comput. **60**(1), 93–105 (2011)
3. A. Alaghi, C. Li, J.P. Hayes, Stochastic circuits for real-time image-processing applications, in *Proceedings of the 50th Annual Design Automation Conference* (ACM, New York, 2013), p. 136
4. S.S. Tehrani, S. Mannor, W.J. Gross, Fully parallel stochastic LDPC decoders. IEEE Trans. Signal Process. **56**(11), 5692–5703 (2008)
5. B.D. Brown, H.C. Card, Stochastic neural computation. II. Soft competitive learning. IEEE Trans. Comput. **50**(9), 906–920 (2001)
6. B.D. Brown, H.C. Card, Stochastic neural computation. I. Computational elements. IEEE Trans. Comput. **50**(9), 891–905 (2001)
7. P. Li, W. Qian, M.D. Riedel, K. Bazargan, D.J. Lilja, The synthesis of linear finite state machine-based stochastic computational elements, in *17th Asia and South Pacific Design Automation Conference* (IEEE, Washington, DC, 2012), pp. 757–762
8. P. Li, D.J. Lilja, W. Qian, K. Bazargan, M. Riedel, The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic, in *Proceedings of the International Conference on Computer-Aided Design* (ACM, New York, 2012), pp. 480–487
9. G. Lorentz, *Bernstein Polynomials* (University of Toronto Press, Toronto, 1953)
10. A. Alaghi, J.P. Hayes, A spectral transform approach to stochastic circuits, in *Computer Design (ICCD), 2012 IEEE 30th International Conference on* (IEEE, Washington, DC, 2012), pp. 315–321
11. T.H. Chen and J.P. Hayes, Equivalence among stochastic logic circuits and its application, in *Proceedings of the 52nd Annual Design Automation Conference* (ACM, New York, 2015), p. 131

12. Z. Zhao and W. Qian, A general design of stochastic circuit and its synthesis, in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (EDA Consortium, San Jose, 2015), pp. 1467–1472
13. D. Baneres, J. Cortadella, and M. Kishinevsky, A recursive paradigm to solve Boolean relations, in *Proceedings of the 41st annual Design Automation Conference* (ACM, New York, 2004), pp. 416–421
14. R.L. Rudell, Multiple-valued logic minimization for PLA synthesis (No. UCB/ERL-M86/65). California Univ Berkeley Electronics Research Lab (1986)

Chapter 10
Decomposition of Index Generation Functions Using a Monte Carlo Method

Tsutomu Sasao and Jon T. Butler

1 Introduction

One of the important tasks in logic synthesis is to find a circuit structure that is suitable for implementation. **Functional decomposition** [1, 4] is a technique to decompose a circuit into two subcircuits each with a lower cost than the original circuit. Various techniques to find functional decompositions have been presented [2, 6, 8, 10]. They are routinely used in logic synthesis. Many circuits that are used in computers have some structure, and they are not random [3]. On the other hand, almost all randomly generated switching functions have no effective decompositions [12].

Recently, we are working on index generation functions [13, 14]: $f : \{0, 1\}^n \rightarrow \{0, 1, 2, \ldots, k\}$, where $k \ll 2^n$. Here, n is the number of bits in each registered vector, and k is the total number of registered vectors.

They are used for access control lists, routers, and virus scanning for the internet, etc. Index generation functions found in practical applications have properties similar to those of randomly generated index generation functions.

In this chapter, we show that index generation functions often have effective decompositions. To show this, we use a Monte Carlo method to predict the column multiplicity of the decomposition charts for random index generation functions.

An index generation function can be efficiently implemented by an LUT or an **IGU** (Index Generation Unit, Fig. 10.1), which are programmable [13]. We omit the

T. Sasao (✉)
Department of Computer Science, Meiji University, Kawasaki 214-8571, Japan
e-mail: sasao@cs.meiji.ac.jp

J.T. Butler
Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA 93943-5121, USA

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_10

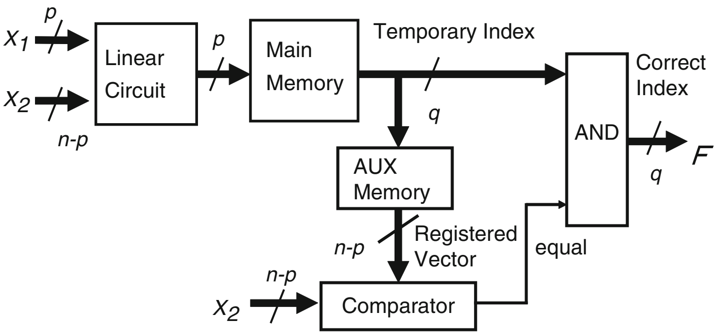


Fig. 10.1 Index generation unit (IGU)

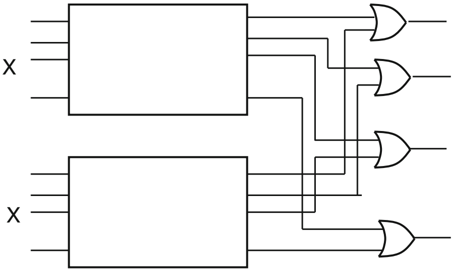


Fig. 10.2 Realization of a function by a parallel decomposition

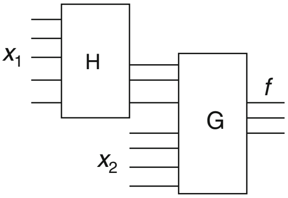


Fig. 10.3 Realization of a logic function by serial decomposition

explanation of the operation of an IGU, which appeared in [14].[1] Currently, an IGU is implemented by a combination of field programmable gate arrays (FPGAs) and memories [9]. However, a single-chip custom LSI can also be used to implement an index generation function. Suppose that LSIs for IGUs with n inputs and weight k are available. For a function with larger k, we can partition the set of vectors into several sets, and implement each by an independent IGU. The outputs of the IGUs can be combined by an OR gate to produce the final output [15]. This is a **parallel decomposition** (Fig. 10.2). On the other hand, for a function with larger n, we can partition the set of input variables into two sets X_1 and X_2 to produce the function. This is a **serial decomposition** (Fig. 10.3).

In our applications, when we prepare the programmable IGU chip, we only know the values of n and k, but do not know the detail of the functions. Functions to be

[1] For simplicity, readers can assume that LUTs are used to implement the functions.

implemented are different for different users. This situation is similar to the case of FPGAs: FPGA companies do not know all the functions to implement in advance. In this chapter, we try to predict the complexity of random functions.

The rest of the chapter is organized as follows: Section 2 introduces functional decomposition. Section 3 introduces the properties of index generation functions. Section 4 shows a Monte Carlo method to derive column multiplicity of decomposition charts for random index generation functions. Section 5 shows a procedure for computing the column multiplicity of decompositions. Section 6 shows the experimental results. Section 7 shows a method to assess the programmable architecture for index generation functions. Section 8 concludes the chapter.

2 Decomposition

In this part, we introduce basic concepts of functional decomposition.

Definition 2.1 ([1, 11]) Let $f(X)$ be a function, and (X_1, X_2) be a partition of the input variables, where $X_1 = (x_1, x_2, \ldots, x_s)$ and $X_2 = (x_{s+1}, x_{s+2}, \ldots, x_n)$. The **decomposition chart** for f is a two-dimensional matrix with 2^s columns and 2^{n-s} rows, where each column and row is labeled by a unique binary assignment of values to the variables. Each assignment maps under f to $\{0, 1, \ldots, k\}$. The function represented by a column is a **column function** and is dependent on X_2. Variables in X_1 are **bound variables**, while variables in X_2 are **free variables**. In the decomposition chart, the **column multiplicity**, denoted by μ, is the number of different column functions.

Example 2.1 Figure 10.4 shows a decomposition chart of a 4-variable switching function. $X_1 = (x_1, x_2)$ denotes the bound variables, and $X_2 = (x_3, x_4)$ denotes the free variables. Since all the column patterns are different and there are four of them, the column multiplicity is $\mu = 4$. ■

Theorem 2.1 ([4]) *For a given function f, let X_1 be the bound variables, let X_2 be the free variables, and let μ be the column multiplicity of the decomposition chart. Then, the function f can be represented as $f(X_1, X_2) = g(h(X_1), X_2)$, and is realized with the network shown in Fig. 10.3. The number of signal lines connecting blocks H and G is $r = \lceil \log_2 \mu \rceil$, where H and G realize h and g, respectively.*

Fig. 10.4 Decomposition chart of an logic function

		0	0	1	1	x_1
		0	1	0	1	x_2
0	0	0	0	0	1	
0	1	1	1	0	0	
1	0	0	1	0	0	
1	1	0	0	0	0	
x_3	x_4					

The logic functions for H and G can be realized by memories, and the complexities for G and H can be measured by the number of bits in the memories. The signal lines connecting H and G are called **rails**. When the number of rails r is smaller than the number of input variables in X_1, it is **support-reducing** [5], and we can often reduce the total amount of memory to realize the logic in Fig. 10.3.

3 Index Generation Functions and Their Properties

Definition 3.1 Consider a set of k different binary vectors of n bits. These vectors are **registered vectors**. For each registered vector, assign a unique integer from 1 to k, called an **index**. A **registered vector table** shows, for each registered vector, its index. An **index generation function** f produces the corresponding index if the input matches a registered vector, and produces 0 otherwise. k is the **weight** of the index generation function. An index generation function represents a mapping: $f : B^n \rightarrow \{0, 1, 2, \ldots, k\}$, where $B = \{0, 1\}$.

Example 3.1 Table 10.1 shows a registered vector table with $k = 4$ vectors. The corresponding index generation function is shown in Table 10.2. In this case, the output is represented by 3 bits. So, it shows a mapping $B^4 \rightarrow \{0, 1, 2, 3, 4\}$. Note that the index values from Table 10.2 are shown in binary in bold. Also, note that registered vectors missing in Table 10.1 are shown in Table 10.2 mapped to 000. ■

Typically, k is much smaller than 2^n, the total number of input combinations.

Example 3.2 Consider the decomposition chart in Fig. 10.5. It shows an index generation function $f(X)$ with weight 7. $X_1 = (x_1, x_2, x_3, x_4)$ denotes the bound variables, and $X_2 = (x_5)$ denotes the free variable. Note that the column multiplicity of this decomposition chart is 7. ■

Lemma 3.1 *Let $\mu(f(X_1, X_2))$ be the column multiplicity of a decomposition chart of an index generation function f, let k be the weight of f, and let s be the number of variables in X_1. Then,*

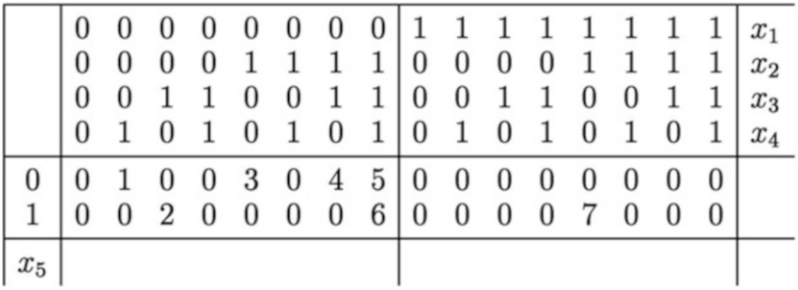
$$\mu(f(X_1, X_2)) \leq \min\{k + 1, 2^s\}.$$

Table 10.1 Registered vector table

Vector				Index
x_1	x_2	x_3	x_4	
1	0	0	1	1
1	1	1	1	2
0	1	0	1	3
1	1	0	0	4

Table 10.2 Index generation function

Input				Output		
x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	**0**	**1**	**1**
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	0	1	**0**	**0**	**1**
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	**1**	**0**	**0**
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	**0**	**1**	**0**



	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_1
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_2
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_4
0	0	1	0	0	3	0	4	5	0	0	0	0	0	0	0	0	
1	0	0	2	0	0	0	0	6	0	0	0	0	7	0	0	0	
x_5																	

Fig. 10.5 Decomposition chart for f

Proof Since the number of non-zero outputs is k, the column multiplicity never exceeds $k + 1$. Further, the column multiplicity never exceeds the total number of columns, 2^s. □

Lemma 3.2 *Let f be an index generation function with weight k. Then, there exists a functional decomposition $f(X_1, X_2) = g(h(X_1), X_2)$, where g and h are index generation functions, such that the weight of g is k, and the weight of h is at most k.*

Proof Consider a decomposition chart, in which X_1 denotes the bound variables, and X_2 denotes the free variables. Let $X_1 = (x_1, x_2, \ldots, x_s)$, where $s \geq \lceil \log_2(k+1) \rceil$.

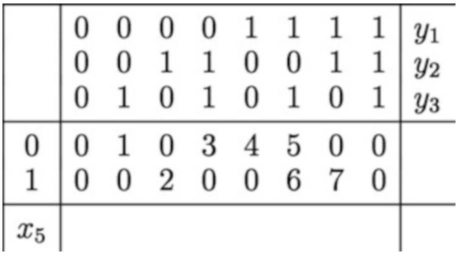
Let h be a function where the variables are X_1, and the output values are defined as follows: Consider the decomposition chart, where assignments of values to X_1 label columns (i.e., bound variables). For the assignments to X_1 corresponding to columns with only zero elements, $h = 0$. For other assignments, the outputs of h are distinct integers from 1 to w_h, where w_h denotes the number of columns that have non-zero element(s). Since $w_h \leq k$, the weight of h is at most k, and the number of output values of h is at most $k + 1$. On the other hand, the function g is obtained from f by reducing some columns that have all zero output in the decomposition chart. Thus, the number of non-zero outputs in g is equal to the number of non-zero outputs in f. Thus, g is also an index generation function with weight k. □

Example 3.3 Consider the decomposition chart in Fig. 10.5. Let the function $f(X)$ be decomposed as $f(X_1, X_2) = g(h(X_1), X_2)$, where $X_1 = (x_1, x_2, x_3, x_4)$, and $X_2 = (x_5)$. Table 10.3 shows the function h. It is a 4-variable 3-output logic function with weight 6. The decomposition chart for the function g is shown in Fig. 10.6. As shown in this example, the functions obtained by decomposing the index generation function f are also index generation functions, and the weights of f and g are 6 and 7, respectively. ■

Table 10.3 Truth table for h

x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	0	0	0
0	1	1	0	1	0	0
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

Fig. 10.6 Decomposition chart for g



	0	0	0	0	1	1	1	1	y_1
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_3
0	0	1	0	3	4	5	0	0	
1	0	0	2	0	0	6	7	0	
x_5									

4 Balls into Bins Model

In this part, we show a method to predict the column multiplicity of a functional decomposition using a balls into bins model.

In Definition 2.1, we specified that the first s variables $x_1, x_2, \ldots, x_s$ are in X_1, and the remaining $n - s$ variables are in X_2. However, in many cases, we can select any s variables for X_1. In this case, the problem of functional decomposition is to partition the variables into two sets, so that the number of rails r between two blocks is minimized. To do this, we want to reduce μ. Note that, in order to reduce the rails between H and G (see Fig. 10.5), we must reduce μ so that it is equal to or less than the smallest power of two.

Given a function table, we have to search $\binom{n}{s}$ combinations, where n is the total number of variables, and s is the number of the bound variables. From Lemma 3.1, we have an upper bound on μ:

$$\mu(f(X_1, X_2)) \leq \min\{2^s, k + 1\}.$$

First, we show an exhaustive approach to find a decomposition.

Example 4.1 Consider the registered vector table shown in Table 10.4. It is a 20-variable random index generation function with weight $k = 20$. We need to find an effective decomposition that reduces the implementation cost. The number of decompositions is equal to the number of ways to partition the set of variables into two non-empty sets, that is $2^n - 2$. If we know the expected column multiplicity, then we can predict how likely exhaustive search can find a good decomposition. Suppose that the number of bound variables is $s = 8$. In this case, the decomposition chart has $2^8 = 256$ columns and $2^{20-8} = 2^{12} = 4096$ rows. Then, the number of decompositions to check is $\binom{20}{8} = 125{,}970$. By exhaustive search, we find the minimum column multiplicity to be $\mu = 15 + 1 = 16$. One decomposition was found with this column multiplicity; it has for the bound variables $X_1 = (x_1, x_2, x_5, x_7, x_{10}, x_{13}, x_{14}, x_{17})$. In this case, the number of rails is reduced to four (from five). ■

An approach in which we generate a random index generation function, cast it into a decomposition chart with one of many choices for the bound variables, compute the column multiplicity, and choose the minimum is computationally too

Table 10.4 Registered vector table for a 20-variable index generation function

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}	f
0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	1
0	0	0	1	1	0	0	0	1	0	0	1	1	0	1	0	1	1	0	0	2
1	0	1	0	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	1	3
0	0	1	1	0	1	0	1	1	0	0	1	1	0	1	0	1	1	1	0	4
1	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1	5
1	1	0	0	1	0	0	1	0	0	1	0	1	0	1	1	0	0	1	1	6
1	0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	0	1	7
1	0	0	0	1	0	1	1	0	0	1	1	0	0	1	0	1	1	1	1	8
0	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	1	0	1	9
0	1	0	1	1	0	0	1	0	1	1	1	0	1	0	0	1	0	0	0	10
1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	1	0	0	11
0	1	0	0	1	0	0	1	1	1	0	1	1	0	1	0	1	1	0	1	12
0	0	0	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	1	0	13
0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	14
0	1	1	1	1	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1	15
1	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1	1	0	1	0	16
0	0	1	1	0	0	0	0	1	0	1	1	1	0	1	1	1	0	0	0	17
1	0	0	1	1	0	0	0	0	0	1	0	1	0	1	1	0	0	1	1	18
0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	0	0	0	19
1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0	1	1	1	1	20

expensive. To estimate the distribution of column multiplicities, we need a more efficient method.

The column multiplicity μ of an index generation function with weight k can be predicted by the **balls into bins model** as follows: In the decomposition chart of an index generation function, all care values occur in exactly one of the columns. Since all care values are distinct, any column with a care value is distinct from all other columns. The only columns that are identical are those that contain no care values. Therefore, the column multiplicity of a decomposition chart is just the number of columns with at least one care values plus 1 if there is at least one column with no care values.

Consider another random process in which there are as many bins as there are *columns* in the decomposition chart. We can distribute k distinct balls into bins with $0,1,\ldots,$ balls allows in each bin. The random distribution of balls in this way is equal to choosing a bin number with repetition, one for each ball. If repetition occurs, then multiple balls fall into the same bin. Assume that there are 2^s distinct bins, and k distinct balls are randomly thrown into these bins. If all the balls fall into the same bin,[2] then $\mu = 1 + 1 = 2$. If all the balls fall into k different bins, then $\mu = \min\{k + 1, 2^s\}$. However, in most cases, $2 < \mu \leq k + 1$. The number of non-empty bins plus one is equal to the column multiplicity μ.

An efficient method to simulate the balls and bin model is the **integer set model** as follows: Assume that a set of k integers represented as standard binary numbers on s bits are randomly generated .

Example 4.2 Let us predict the column multiplicity for random index generation functions of $n = 20$ variables and weight $k = 20$ by Monte Carlo simulation using the integer set model. The number of different n variable index generation functions with weight k is $P(2^n, k) = \frac{2^n!}{(2^n-k)!}$. For $n = 20$ and $k = 20$, this is about 2.6×10^{120}, which is too large to search exhaustively.

We can efficiently approximate this approach as follows. Generate uniformly distributed $k = 20$ binary numbers each with $s = 8$ bits. The eight bits represent a value from 0 through 255, and correspond to a bin number. The first 8-bit number is the bin number associated with an index 1, the second is the bin number associated with an index of 2, etc. Table 10.5 shows the result of this Monte Carlo simulation using the integer set model. In this case, we generated 10^6 samples. ■

Table 10.5 shows that, in most cases, the number of distinct integers is either 21 or 20. However, this value can be reduced to $\mu = 14$ for one case, $\mu = 15$ for 20 cases, and $\mu = 16$ for 297 cases. This implies that the functions have non-zero probabilities with decompositions where $\mu \leq 16$. However, the probability of a decomposition with $\mu \leq 13$ is quite low, since a sample set of size 10^6 failed to produce a single decomposition with $\mu \leq 13$. ■

We assume that the functions that appear in the search of the decompositions, and the random set of integers used in a Monte Carlo simulation have similar

[2] This assumes that $k \leq 2^{n-s}$.

Table 10.5 Number of distinct ($s = 8$)-bit integers

Rails (r)	Number of distinct integers (μ)	Occurrence
4	$13 + 1$	1
4	$14 + 1$	20
4	$15 + 1$	297
5	$16 + 1$	3229
5	$17 + 1$	25,749
5	$18 + 1$	129,422
5	$19 + 1$	374,505
5	$20 + 1$	466,777

statistical properties. That is, the probability distribution functions for the functional decompositions are similar to that of random integer sets. The validity of this assumption will be checked in the experiments in Sect. 6.

5 Procedure to Compute the Column Multiplicity

To validate the use of a Monte Carlo technique in estimating the column multiplicity, we compare the results obtained by a Monte Carlo technique with an exact enumeration in which we enumerate all binary arrays according to the column multiplicity. Every binary array is enumerated exactly once, and, as such, it can be considered a proxy for a "perfect" Monte Carlo simulation. We use the balls into bins model.

Lemma 5.1 *Assume that there are t non-distinct bins and k distinct balls. The number of different ways to put k distinct balls into t non-distinct bins is*

$$S(k,t) = \begin{cases} 1, & \text{if } (t = 1 \; or \; t = k) \\ S(k-1,t-1) + tS(k-1,t), & \text{otherwise.} \end{cases}$$

Proof The proof is done by a mathematical induction. $S(k,t)$ can be calculated for three cases:

When $t = 1$: All the balls are in one bin. So, there is only one way.
When $t = k$: All the balls are in k bins. So, there is only one way.

Otherwise: Assume that $k - 1$ balls are already in the bins, and the k-th ball is put into one of the bins. In this case, there exist two cases:

Table 10.6 Values for $S(k, t)$

	t								
k	1	2	3	4	5	6	7	8	9
1	1								
2	1	1							
3	1	3	1						
4	1	7	6	1					
5	1	15	25	10	1				
6	1	31	90	65	15	1			
7	1	63	301	350	140	21	1		
8	1	127	966	1701	1050	266	28	1	
9	1	255	3025	7770	6951	2646	462	36	1
10	1	511	9330	34,105	42,525	22,827	5880	750	45

1. The $k-1$ balls are in $t-1$ bins, but one bin is empty. In this case, the number of ways to put the first $k-1$ balls into $t-1$ bins is $S(k-1, t-1)$, by the hypothesis. The k-th ball is put into the empty bin.
2. The $k-1$ balls are in t bins. No bin is empty. In this case, the number of ways to put the first $k-1$ balls into t bins is $S(k-1, t)$, by the hypothesis. Also, there are t ways to put the k-th ball into one of t bins. So, the total number of ways is $tS(k-1, t)$. From these, we have the lemma.

□

Note that $S(k, t)$ is the **Stirling number of the second kind** [7].

Example 5.1 Table 10.6 shows the values of $S(k, t)$ for $k \leq 10$ and $t \leq 9$.

Theorem 5.1 *Consider the binary arrays that have 2^s columns and k rows. The number of arrays with t distinct columns is*

$$c_{k,t} = P(2^s, t)S(k, t),$$

where $S(k, t)$ is the Stirling number of the second kind, and $P(n, r) = \frac{n!}{(n-r)!}$.

Proof The number of ways to permute t distinct patterns out of 2^s distinct patterns is $P(2^s, t)$. □

To validate the Monte Carlo technique, we ran the simulation for $s = 3$ and $k = 8$ for a total of $(2^s)^k = 2^{24} = 16{,}777{,}216$ samples, which is the number of 3 × 8 binary arrays. We also used the exact enumeration using Theorem 5.1. Table 10.7 compares the results. This shows that there is a close correlation between the Monte Carlo simulation and exact enumeration.

Table 10.7 Comparison of the Monte Carlo technique with exact enumeration $(s = 3, k = 8)$

# of distinct columns	Monte Carlo	Theorem 5.1
1	4	8
2	7162	7112
3	326,013	324,576
4	2,857,425	2,857,680
5	7,053,129	7,056,000
6	5,362,312	5,362,560
7	1,130,883	1,128,960
8	40,288	40,320
Total	16,777,216	16,777,216

For small k, we can pre-compute the table of the Stirling numbers, and store it in the hard disk to compute $c_{k,t}$. However, for large k, the table becomes too large. Thus, we use the Monte Carlo approach for large k.

6 Experimental Results

6.1 Decompositions of 20-Variable Functions

We assume that the distribution of the column multiplicities during decompositions is similar to the random integer sets used in the Monte Carlo simulation.

To confirm this, we generated ten sample index generation functions of $n = 20$ and $k = 20$. Then, for each function, we counted the column multiplicities for all the decompositions, where the number of bound variables is $s = 8$. Note that the number of ways to select 8 variables out of 20 variables is $\binom{20}{8} = 125{,}970$.

Table 10.8 summarizes the distributions of column multiplicities. The first column shows the number of rails $r = \lceil \log_2 \mu \rceil$. The second column shows the column multiplicity μ. The third to 12th columns show the distributions for f_1 to f_{10}. For example, in f_1, the minimum column multiplicity is $\mu = 15 + 1$, and 20 decompositions produce this μ. For f_5, the minimum multiplicity is $\mu = 13+1$, and 10 decompositions produce this μ. For f_9, the minimum multiplicity is $\mu = 14 + 1$, and 18 decompositions produce this μ. For the other 8 functions, the minimum multiplicities are $\mu = 15 + 1$. The column headed with *SUM* denotes the sum of 10 sample functions. The rightmost column, headed with *Monte* denotes the result of the Monte Carlo simulation using the integer set model. The number of sample sets generated for the simulation is $10 \times \binom{20}{8} = 1{,}259{,}700$, which is equal to the total number of decompositions for 10 sample functions.

Table 10.8 Decompositions of 20-variable index generation functions with $k = 20$ ($s = 8$)

r	μ	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	*SUM*	*Monte*
4	13+1	0	0	0	0	10	0	0	0	0	0	10	2
4	14+1	0	0	0	0	181	0	0	0	18	0	199	25
4	15+1	20	18	27	7	1338	5	24	3	138	9	1589	374
5	16+1	457	571	353	164	5478	144	366	122	1237	172	9064	4101
5	17+1	4673	4775	3346	1669	14,934	1761	2645	1683	6413	2242	44,141	32,505
5	18+1	22,084	20,615	18,172	10,841	28,903	11,668	14,770	13,178	23,163	13,887	177,281	163,015
5	19+1	48,736	49,356	49,694	41,775	40,076	42,141	46,962	47,846	49,122	46,922	462,630	471,686
5	20+1	50,000	50,635	54,378	71,514	35,050	70,251	61,203	63,138	45,879	62,738	564,786	587,992

Table 10.8 shows that, for all 10 sample functions, the column multiplicities can be reduced to $\mu = 15 + 1$ or less. This means that the number of rails r can be reduced from five to four. The Monte Carlo simulation shows that, for $374 + 25 + 2 = 401$ combinations out of 1,259,700, the column multiplicities are reduced to $\mu = 15 + 1$ or less. This shows that there is an incentive to find a decomposition with small column multiplicity, but it may be hard to find.

6.2 Decomposition of a 64-Variable Function

In the previous experiment, the number of variables was only 20, and the number of the decompositions was $\binom{20}{8} = 125{,}970$.

In this part, we used a sample function with $n = 64$ and $k = 20$. As before, the number of bound variables is $s = 8$. Note that, the number of decompositions is now $\binom{64}{8} = 4{,}426{,}165{,}368$. Table 10.9 shows the results. The first column shows the number of rails r; the second column shows the column multiplicities μ; the third column shows the number of decompositions that produced the corresponding μ. The rightmost column headed by *Monte* shows the number of occurrences in the Monte Carlo simulation. In the Monte Carlo simulation, the number of possible ways to generate sets of 20 integers of 8 bits is $(2^8)^{20} = 2^{160} \simeq 1.46 \times 10^{48}$. In this experiment, we generated $\binom{64}{8} = 4{,}426{,}165{,}368$ sample sets.

In this example, the Monte Carlo method provides a good approximation when μ is large, but not so good when μ is small. Note that the CPU time for the Monte Carlo simulation was about 10 min, while that for the exhaustive decomposition search was 22 min.

Table 10.9 Decomposition of a 64-variable index generation function with $k = 20$ ($s = 8$)

r	μ	Occurrence	*Monte*
4	$11 + 1$	0	1
4	$12 + 1$	20	73
4	$13 + 1$	971	2768
4	$14 + 1$	33,301	69,099
4	$15 + 1$	759,722	1,196,778
5	$16 + 1$	11,265,390	14,274,472
5	$17 + 1$	103,232,971	113,605,327
5	$18 + 1$	562,995,509	574,201,989
5	$19 + 1$	1,675,277,777	1,656,555,761
5	$20 + 1$	2,072,599,707	2,066,259,100
	Total	4,426,165,368	4,426,165,368

With the Monte Carlo simulation, we can see that the probability of finding a decomposition with $\mu = 16$ (i.e., $r = 4$ rails) is quite high if the good solutions are distributed uniformly, and if we try more than 10^4 samples randomly. However, the probability that with $\mu \leq 8$ (i.e., $r = 3$ rails) is almost zero. Thus, once we find a decomposition with $\mu \leq 16$, we can stop the search; it is not likely that we will find a decomposition with a smaller r.

7 A Method to Assess Programmable Architecture

Problem 1 Design a programmable architecture for an index generation function with $n = 500$ and $k = 100$ using a pair IGUs.

(Solution) In a decomposition, the number of rails is at most $q = \lceil \log_2(k+1) \rceil = 7$. A Monte Carlo simulation with $s = 11$ and $k = 100$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 87$. Since the number of rails is $r = \lceil \log_2 \mu \rceil = 7$, the function can be realized by a cascade as shown in Fig. 10.7. IGU1 has 255 inputs and 7 outputs, while IGU2 has $7 + 245 = 252$ inputs and 7 outputs.

Problem 2 Design a programmable architecture for an index generation function with $n = 20$ and $k = 68$ using a pair of LUTs.

(Solution) In a decomposition, the number of rails is at most $q = \lceil \log_2(k+1) \rceil = 7$, by Lemma 3.1. Let the number of bound variables be $s = 10$. The Monte Carlo simulation with $s = 10$ and $k = 68$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 57$. Thus, the number of rails is reduced to $\lceil \log_2 \mu \rceil = 6$. The function can be realized by a cascade as shown in Fig. 10.8. LUT1 has 10 inputs and 6 outputs, while LUT2 has $6 + 10 = 16$ inputs and 7 outputs. Since LUT2 has 16 inputs, and is much larger than LUT1, the circuit is not so efficient.

So, we increase the number of inputs to LUT1 to $s = 12$. A Monte Carlo simulation with $s = 12$ and $k = 68$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 62$. Thus, the number of rails is

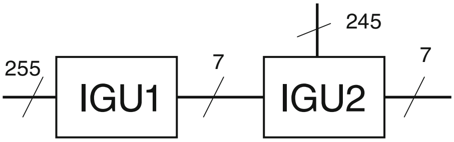


Fig. 10.7 Architecture using two IGUs ($n = 50$ and $k = 100$)

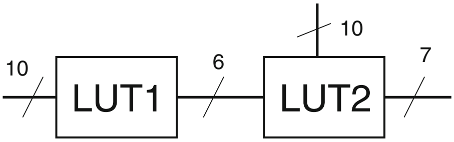


Fig. 10.8 Architecture using two LUTs ($n = 20$, $s = 10$ and $k = 68$)

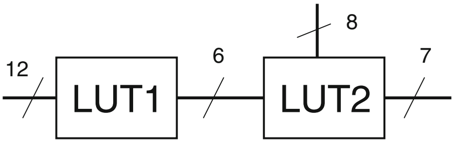


Fig. 10.9 Architecture using two LUTs ($n = 20$, $s = 12$ and $k = 68$)

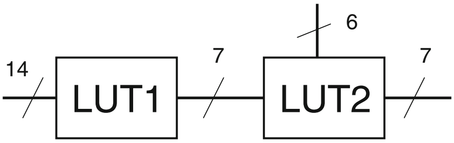


Fig. 10.10 Architecture using two LUTs ($n = 20$, $s = 14$ and $k = 68$)

still $\lceil \log_2 \mu \rceil = 6$. The function can be realized by a cascade as shown in Fig. 10.9, which is more efficient than Fig. 10.8.

Next, we further increase the number of inputs to LUT1 to $s = 14$. A Monte Carlo simulation with $s = 14$ and $k = 68$ shows that the minimum column multiplicity of the decompositions among 10^6 samples is $\mu = 65$. Thus, the number of rails is increased to $\lceil \log_2 \mu \rceil = 7$, as shown in Fig. 10.10, which is less efficient than Fig. 10.9.

8 Conclusion and Comments

In this chapter, we present a Monte Carlo method to predict the column multiplicity of the decomposition charts for random index generation functions. We also show a procedure to compute the multiplicities of decomposition charts. Comparison with the exact enumerations shows that the Monte Carlo method using integer model produces good approximations to exact enumeration.

When we design a programmable architecture for index generation functions, in many cases, we know only the numbers of inputs and registered vectors, but not the detail of the functions. In such cases, the method to assess the programmable architecture presented in this chapter is quite useful.

A different, but related problem is to find decompositions for specific index generation functions. We developed a heuristic [16] and exact [17] algorithms to find decompositions of index generation functions. In a functional decomposition algorithm, when a decomposition with a column multiplicity μ_1 is found, the next step is to find a decomposition with a column multiplicity μ_2, where $\lceil \log_2 \mu_2 \rceil < \lceil \log_2 \mu_1 \rceil$. If there is no solution, we can stop the search. Analysis of column multiplicities for index generation functions was useful to find these algorithms.

Acknowledgements This research is partly supported by the Japan Society for the Promotion of Science (JSPS) Grant in Aid for Scientific Research. Discussion with Mr. Kyu Matsuura was useful to improve Sect. 5. Also, the reviewers' comments improved the presentation of the chapter.

References

1. R.L. Ashenhurst, The decomposition of switching functions, in *International Symposium on the Theory of Switching*, April 1957, pp. 74–116
2. V. Bertacco, M. Damiani, The disjunctive decomposition of logic functions, in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD-1997)*, November 1997, pp. 78–82
3. R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic, Boston, 1984)
4. H.A. Curtis, *A New Approach to the Design of Switching Circuits* (D. Van Nostrand Co., Princeton, NJ, 1962)
5. V. Kravets, K. Sakallah, Constructive library-aware synthesis using symmetries, in *Design, Automation and Test in Europe (DATE-2000)*, Paris, March 2000, pp. 208–213
6. Y-T. Lai, M. Pedram, S.B.K. Vrudhula, BDD based decomposition of logic functions with application to FPGA synthesis, in *30th ACM/IEEE Design Automation Conference (DAC-1993)*, June 1993, pp. 642–647
7. C.L. Liu, *Introduction to Combinatorial Mathematics* (McGraw-Hill, New York, 1968)
8. Y. Matsunaga, An exact and efficient algorithm for disjunctive decomposition, in *Synthesis and System Integration of Mixed Technologies (SASIMI-1998)*, October 1998, pp. 44–50
9. H. Nakahara, T. Sasao, M. Matsuura, H. Iwamoto, Y. Terao, A memory-based IPv6 lookup architecture using parallel index generation units. IEICE Trans. Inf. Syst. **E98-D**(2), 262–271 (2015)
10. T. Sasao, FPGA design by generalized functional decomposition, in *Logic Synthesis and Optimization*, ed. by T. Sasao (Kluwer Academic, Dordrecht, 1993), pp. 233–258
11. T. Sasao, *Switching Theory for Logic Synthesis* (Kluwer Academic, Dordrecht, 1999)
12. T. Sasao, Totally undecomposable functions: applications to efficient multiple-valued decompositions, in *International Symposium on Multiple-Valued Logic (ISMVL-1999)*, Freiburg, 20–23 May 1999, pp. 59–65
13. T. Sasao, *Memory-Based Logic Synthesis* (Springer, New York, 2011)
14. T. Sasao, Index generation functions: tutorial. J. Mult. Valued Log. Soft Comput. **23**(3–4), 235–263 (2014)
15. T. Sasao, A realization of index generation functions using multiple IGUs, in *International Symposium on Multiple-Valued Logic (ISMVL-2016)*, Sapporo, 17–19 May 2016, pp.113–118
16. T. Sasao, K. Matsuura, Y. Iguchi, A heuristic decomposition of index generation functions with many variables, in *The 20th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI-2016)*, Kyoto, 24 October 2016, R1–6, pp. 23–28
17. T. Sasao, K. Matsuura, Y. Iguchi, An algorithm to find optimum support-reducing decompositions for index generation functions, in *Design, Automation and Test in Europe (DATE-2017)*, Lausanne, 27–31 March 2017

Index

A
ABC tool, 180–181
Active nodes, 23–25, 32
Amorphous data-parallelism, 22, 26
AND gate, 189, 190
And-inverter graph (AIG), 87, 88, 91, 181
Automated conversion, 126
Autonomous scheduling, 28, 38

B
Back-end-of-line (BEOL), 92, 93, 97
BDD-based algorithms, 170
Bellman-Ford algorithm, 23, 25, 26, 28
BINARY, 175
Blazegraph, 9
Boolean functions, 104, 105, 170, 190
Boolean operators, 52
Branch and Merge operators, 130, 142
Breadth-first search (BFS), 33
Bulk-Synchronous Parallel (BSP-style) semantics, 27

C
Canonical representation, 173
Coarray FORTRAN, 21
Coloured Petri Nets (CPN), 126, 148
Common sub-expressions (CSE), 53, 54
Compressed-sparse-row (CSR) representation, 29
Connected components (CC), 33, 34
Coordinated scheduling, 28

D
Data-driven algorithms, 25–26
Data-mining algorithms, 22
DeepQA software architecture, 15, 16, 18
Delaunay mesh refinement, 31, 32
Dependency quantified Boolean formulas (DQBFs)
  And-Inverter graph (AIG), 163
  benchmarks dealing, 166
  CNF transformation, 162
  Craig interpolants, 163
  elimination sets, 161–162
  iDQ, 163–164
  prototypic solvers, 166
  quantifier prefix, 160
  SAT problem, 160
  semantics of, 153
  Skolem functions, 152, 163
  solver-based approaches, 152
  solvers, scalability of, 167
  syntax of, 153
  unit and pure literals, 161
Design flow
  disadvantages, 75
  logic synthesis frontend, 74–75
  physical awareness (*see* Physical awareness, design flow)
  physical design backend, 75
  VLSI (*see* VLSI design)
Digital microfluidic biochips (DMFBs)
  basic unit cell, 44, 45
  design methods
    automated synthesis of, 45, 46

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3

Digital microfluidic biochips (DMFBs) (*cont.*)
cyberphysical DMFBs, 47
epigenetics, 46
5-layer (C^5) architecture for, 46, 47
quantitative analysis of gene expression, 48, 49
shared-resource coordination framework, 48, 50
Dijkstra’s algorithm, 23–25
DMFBs. *See* Digital microfluidic biochips (DMFBs)

E
Eager Fork operator, 127
EDA tools, 125
Elastic coarse Grain Reconfigurable Arrays (CGRAs), 126
Elastic systems, 125
Electronic design automation (EDA), 69
applications, 10–11
10B+ transistor processor, 22 nm technology, 7
DeepQA architecture, 15, 16, 18
estimated storage capacity requirements, 7, 8
Google maps, 8, 9
hierarchical design process, 14
IBM Power8 Linux server, 11
lack of prediction, design flow, 13
Neo4j, 11
noise inspection tool, 11, 12
reducing structural bias in technology mapping, 15, 17
synthesis parameters, impact of, 13, 14
wavefront technology mapping, 15, 16
big data and analytics, 2
challenges, 2, 10
EDA1.0, 1
EDA2.0, 2
EDA3.0, 2, 3, 18
IT, 3, 18
parallel programming, EDA tools
Galois system (*see* Galois system)
graph algorithms, abstractions for (*see* Graph algorithms)
warehouse-scale computing
application-level software, 4, 6
cluster-level infrastructure, 3–6
core design data, 4
derived data, 4
GraphX stack, 5
platform-level software, 3, 4
ESPRESSO-style algorithms, 170

F
FabScalar-2W OoO core, 142
Field programmable gate arrays (FPGAs), 37–38
Floating Point Unit (FPU), 126
Fluid Pipelines
benchmarks, 142–143
channel groups, 134, 136–137
Coloured Petri Nets, 148
communication and flow control
Branch and Merge operator, 130
control operators, 129
data-dependent selection signal, 130
disjoint or-causality, 130
elastic buffers (EBs) implementation, 129
Fork and Join operators, 130
multiple input/output channels., 129
or-firing rule, 130
deadlock avoidance
Fork operator, 134, 135
no-extraneous dependencies (NED), 132, 133
pseudo-verilog implementation, 133, 134
self-cleaning (SC), 132, 133
design space and energy-delay trade-offs, 149
digital design, 137, 139
elastic FPU, 145–146
elastic OoO core, 146–148
elastic system approaches, 128
energy-delay curve, 145
evaluation methodology
Coloured Petri Nets (CPN), 140
firing semantics, 140
timing, 140–141
FabScalar-2W OoO core, 142, 143
FPU block diagram, 142
logic-oriented design methodology, 149
Pareto frontier, 144
recycling, 143–144
Register Files (RF), 137–138
repipelining, 131–132
SELF and and LI-BDNs, 142
2-way Out-of-Order FabScalar core, 142
Fork and Join operators, 126, 130
FPU block diagram, 142
Functional isomorphism, 104, 105

G

- Galois system, 28–30
  - compressed-sparse-row representation, 29
  - graph analytics
    - Ligra, 32–34
    - PowerGraph, 32–34
  - iterations, 29–30
  - large-scale shared-memory machines, 31–32
  - maze routing in FPGAs, 37–38
  - push-style BFS, 29
  - subgraph isomorphism
    - data graphs for, 36
    - definition, 34
    - generic graph query algorithm, 34, 35
    - query graph for, 36
    - Ullmann algorithm, 35–37
    - VF2 algorithm, 36, 37
  - unordered algorithms, 30
- Gate-level circuits, transparent logic. *See* Transparent logic
- Google maps, 8–10
- Graph algorithms
  - operator formulation
    - global view of algorithms, 25
    - local view of algorithms, 23–25
  - parallelism in, 26–27
    - BSP-style semantics, 27
    - Galois system (*see* Galois system)
    - transactional semantics, 27–28
  - TAO analysis, 23, 24, 26
  - topology-driven and data-driven algorithms, 25–26
- GraphX stack, 5
- Grover's search algorithm, 57

H

- Hardware Model Checking Competition 2014, 118, 120, 122
- High Level Synthesis (HLS), 126
- High Performance FORTRAN (HPF), 21

I

- IBM Power8 Linux server, 11
- Incomplete circuits
  - combinational circuits, 155–156
  - sequential circuits
    - bit stream bit$_2$ and bit$_1$, 158
    - black boxes, interface of, 159
    - realizability problem for incomplete sequential circuits (RISC), 156–157
    - Skolem functions, 160
- Index generation functions, Monte Carlo method
  - column multiplicity
    - balls into bins model, 218
    - binary array, 218
    - and exact enumeration, 219–220
    - stirling number of the second kind, 219
    - 64-variable functions, 222–223
    - 20-variable functions, 220–222
  - decomposition
    - bound variables, 211
    - column function, 211
    - subitem column multiplicity, 211
    - decomposition chart, 212–213, 215
    - free variables, 211
    - functional decompositions, balls into bins model, 215–218
    - parallel decomposition, 210
    - rails, 212
    - serial decomposition, 210
  - field programmable gate arrays (FPGAs) and memories, 210
  - heuristic and exact algorithm, 224
  - programmable architecture, 223–224
  - registered vectors, 212
  - registered vector table, 212
  - truth table, 214
- Integer set model, 217
- Integrated photonic circuits, 48
  - photonic logic circuit model
    - cross-bar switch model, 51
    - Mach-Zehnder interferometer, 50–52
    - ring resonator modulator, 51
    - virtual gates, 52, 53
    - XOR-based CSE, 53, 54
  - Si-photonic integration
    - active tuning, 55
    - DC-bias based tuning, 55
    - microheater-based tuning, 55
    - OEICs, 54
    - ONoC, 54, 55
- International Symposium on Physical Design (ISPD), 69
- International Workshop on Logic Synthesis (IWLS), 69, 75
- Irredundant Sum-of-Product (ISOP) algorithm, 171

J

- Just-in-time parallelization, 28

K

KL-cuts
- I/O pins, 91
- KL-cut PAIGs, 88, 92
- logic computation KL-cuts
  - logic synthesis of, 95
  - partial SDCs for, 93, 94
  - physical design of, 95, 96
  - placement of, 92
- physical synthesis of, 98

L

Label computation operators, 24
Lexicographic satisfiability (LEXSAT), 175–176
LT-DEC benchmarks, 181

M

Machine-learning algorithms, 22
Mach-Zehnder interferometers (MZIs), 48, 50–52, 54
Merge operation, 126
Middle-end-of-line (MEOL), 92, 93, 97, 98
MiniSAT, 180
Miniterms
- expansion in cubes
  - canonical expansion, 176–177
  - fast non-canonical expansion, 178
  - greedy canonical cube expansion, 177–178
- generation of
  - canonical SOP, 175–176
  - non-canonical SOP, 175

Monte Carlo method. *See* Index generation functions, Monte Carlo method
Moore's law, 1, 43–44
Morph operators, 24
Multiplexers (MUX), 103, 106, 108, 113

N

Negation-Permutation-Negation (NPN) isomorphism
- composition of, 106, 117
- definition, 105
- transparent word, 107, 110

Neo4j, 11
No-extraneous dependencies (NED), 132, 133
Non-linear delay model (NLDM), 72

O

OpenAcces(OA), 5
OpenCores, 139
Open Source, 6
OpenStack software, 4
Operator formulation of algorithms
- global view of algorithms, 25
- local view of algorithms
  - active node, 23, 24
  - label computation operators, 24
  - memory model, 25
  - morph operators, 24
  - pull-style operator, 24–25
  - push-style operator, 25
  - relaxation operator, 24

Optical networks-on-chip (ONoC), 54, 55
Optimal energy-delay (ED) point, 126
Opto-electronic integrated circuits (OEICs), 54
Ordering, 25
Out-of-order (OoO) execution, 126

P

Pagerank (PR), 33
Pareto frontier, 126, 144
Partial equivalence checking problem (PEC), 155–156
PathFinder, 37
Photonic logic circuit model
- cross-bar switch model, 51
- Mach-Zehnder interferometer, 50–52
- ring resonator modulator, 51
- virtual gates, 52, 53
- XOR-based CSE, 53, 54

Physical awareness, design flow
- cuts for signal distribution and logic computation, 88–89
- global signal distribution 1L-cuts, 92–94
- input data for, 90
- KL-cuts (*see* KL-cuts)
- local SDCs, 89
- PAIGs, 87
- placement of interface pins, 90
- timing closure, sinal distribution and logic computation, 96–98

Placed and-inverter graphs (PAIGs), 87, 88
PowerGraph, 32–34
Power-law graphs, 26

Q

Quantified Boolean formulas (QBFs), 151, 160
Quantum computation, 57
Quine-McCluskey algorithm, 170

R

Random graphs, 26
Register Files (RF), 137–138
Register-transfer level (RTL), 74, 90
Reverse engineering, 105, 115–117
Reversible circuits
  circuit model, 56
  design of, 59–61
  encoder design, 57–58
  power consumption, CMOS generations, 58, 59
  quantum computation, 57
Ring resonators (RRs), 48, 50, 51, 54, 55
Routing resource graph (RRG), 37
Runtime parallelization, 28

S

Self-cleaning (SC), 132, 133
SGI Ultraviolet, 31, 32
Single-source shortest-paths (SSSP), 22, 24, 33
Si-photonics
  active tuning, 55
  advancements in, 49
  DC-bias based tuning, 55
  microheater-based tuning, 55
  OEICs, 54
  ONoC, 54, 55
Software Dataflow Networks, 126
Solver-based techniques, 151
Speed-up techniques
  optimal partial solution, 203
  removal of branches, cube vectors, 202–203
  update count and explored nodes, limitation of, 203–204
Static parallelization, 28
Static timing analysis, 5
Stochastic computing (SC)
  advantages, 189
  applications, 189
  Bernstein polynomial expansion, 190
  Boolean functions, 190
  branch-and-bound algorithm, 200–202
  circuit synthesis problem
    Boolean relation minimization problem, 195
    sum-of-product (SOP), 194
  digital circuits, 189
  experiment results, 204–205
  AND gate, 189, 190
  general form and computation, 191–192
  general function, synthesis of, 192–194
  proposed algorithm
    capacity constraint, 198
    cube vector, 198–200
    disjointness constraint, 198
    multivariate polynomials, 196
    preliminaries, 196–198
    univariate polynomials, 196
  speed-up techniques
    optimal partial solution, 203
    removal of branches, cube vectors, 202–203
    update count and explored nodes, limitation of, 203–204
  stochastic bit streams, 189
  stochastic circuits, 190
Structural isomorphism, 105
Sum of products (SOP), 194
  BDD-based algorithms, 170
  Boolean functions, 171–173
  Boolean satisfiability, 173–174
  canonical SOPs, 171
  ESPRESSO-style algorithms, 170
  experimental setup, 180–181
  global circuit restructuring, 170
  logic optimization, 169
  multi-level logic synthesis, 169
  practical applications, 171
  Quine-McCluskey algorithm, 170
  SAT-based algorithm
    vs. BDD-based SOP generation, 181–184
    miniterms generation, 174–176
    minterms expansion, in cubes, 176–178
    multi-level implementation, 184–186
    multi-output circuits, 174
    on-set and off-set, 174
    redundant cubes, removal of, 179–180
    runtime improvement, 179–180
    single-output function, 174

T

Toffoli circuit, 56
Toffoli gates, 56
Topology-driven algorithms, 25–26
Transactional semantics, 27–28
Transparent logic, 103
  challenges
    ambiguity of transparency, 116, 117
    generalized transparent words, 115–116
    limitations of proposed algorithms, 117
  composition of transparency, 108–109
  experimental results, 117–118
    functional and structural approaches, 119–121
    on unrolled circuits, 121–123

Transparent logic (*cont.*)
  propagation of transparency, 109–110
  transparency identification
    compound word, four depth-one words, 113, 114
    disjoint transparent blocks, 115
    find transparency with given controls, 111–112
    overall algorithm flow, 112–113
    proceeding words, 114
  transparent words, 106–108
Twitter graph, 33

U
Ullmann algorithm, 35–37
Uniform-degree graphs, 26
Unordered algorithms, 25
Unstructured graphs, 21

V
VF2 algorithm, 36, 37
Virtual gate (VG)
  Boolean operators in, 52
  in common subexpression sharing, 53, 54
  complex functions, 53
  implementation, 52
VLSI design
  design constraints, 71
  design convergence and delay information stability, 73
  sources of delay
    cell delay estimation, 72
    delay from late arrival, 71–72
    wire delay estimation, 72
  synthesis tasks, 75
    AND/OR/XOR trees, balancing and unbalancing of, 83, 84
    buffer deletion, 81
    buffering long nets, 80
    buffering nets to reduce fanout, 81, 82
    cell movement, 78–79
    cloning, 83
    composition/decomposition, 84–85
    gate sizing/repowering, 76–77
    inverter absorption, 85–87
    layer assignment, 79–80
    MUX decomposition, 85, 86
    pin swapping, 81, 82
    potential for improvement, 87
    Vt swapping, 77–78
  timing budget, 73
  timing closure, 72–73
  typical VLSI design flow, 74
Voltages (Vt) swapping, 77–78

W
Warehouse-scale computing, EDA
  application-level software, 4, 6
  cluster-level infrastructure, 3–6
  platform-level software, 3, 4

Y
YouTube, 9

Z
Zero-slack algorithm (ZSA), 73