A Study on the Impact of Compiler Optimizations on High-Level Synthesis

Jason Cong, Bin Liu, Raghu Prabhakar, and Peng Zhang

University of California, Los Angeles
{cong,bliu,raghu,pengzh}@cs.ucla.edu

Abstract. High-level synthesis is a design process that takes an untimed, behavioral description in a high-level language like C and produces register-transfer-level (RTL) code that implements the same behavior in hardware. In this design flow, the quality of the generated RTL is greatly influenced by the high-level description of the language. Hence it follows that both source-level and IR-level compiler optimizations could either improve or hurt the quality of the generated RTL. The problem of ordering compiler optimization passes, also known as the phase-ordering problem, has been an area of active research over the past decade. In this paper, we explore the effects of both source-level and IR optimizations and phase ordering on high-level synthesis. The parameters of the generated RTL are very sensitive to high-level optimizations. We study three commonly used source-level optimizations in isolation and then propose simple yet effective heuristics to apply them to obtain a reasonable latency-area tradeoff. We also study the phase-ordering problem for IR-level optimizations from a HLS perspective and compare it to a CPU-based setting. Our initial results show that an input-specific order can achieve a significant reduction in the latency of the generated RTL, and opens up this technology for future research.

Keywords: Compiler Optimization, Design space exploration, High-level synthesis, Phase ordering.

1 Introduction

The field of compiler optimizations has been an area of active research for more than fifty years. Numerous optimizations have been proposed and deployed over the course of time, each trying to optimize a certain aspect of an input program. Optimizations play a key role in evaluating a compiler.

A well-known fact in literature [17] is that optimizations have enabling and disabling interactions among themselves, and the best order is dependent on the program, target and the optimization function. As the solution space is huge, compiler researchers have tried a plethora of methods over the past decade based on searching techniques ([7] [13] [12], [3]), analytical models ([22], [25], [20], [24]), empirical approaches based on statistical data ([19], [18] [2]), and a mixture of all of these ([9], [21], [16]). However, it is to be noted that all aforementioned

H. Kasahara and K. Kimura (Eds.): LCPC 2012, LNCS 7760, pp. 143–157, 2013.
© Springer-Verlag Berlin Heidelberg 2013

approaches have been used in a CPU-based setting. In this case, decisions regarding optimization orders are implicitly or explicitly influenced by execution parameters such as the processor pipeline, size of the instruction window, presence of hardware-managed caches etc. How different would such optimization orders be if the code being optimized was not going to be 'executed' on a processor, but is a behavioral description to be synthesized into some customized hardware itself?

```
int add(int a[20], int o[2])
{
    int i;
    o[0] = 0;
    o[1] = 1;
    for(i = 1; i < 20; i++) {
        if(a[i]%2 == 0) {
            o[0] += a[i];
            o[1] *= a[i];
        }
    }
}
```

(a)

Sequence	CPU(cycles)	HLS (cycles)
gim	3903	12
img	3814	22

(b)

Fig. 1. (a) Example design. (b) CPU vs. HLS setting. CPU best sequence differs from HLS best sequence.

Consider the simple design in Fig. 1. Also, let us consider a set of three optimizations[1]: *global value numbering* (g), *memory to register promotion* (m) and *induction variable canonicalization* (i). The table in Fig. 1(b) summarizes the performance of two sequences *gim* and *img*. The CPU numbers were obtained using Simics, an out-of-order processor simulator, while HLS numbers were obtained using xPilot [4], a research tool for high-level synthesis. We can clearly observe that the sequence *gim* wins in the HLS setting while sequence *img* wins in the CPU setting. We find that *img* produces smaller code with fewer loads, because '*g*' applied after '*m*' is exposed to a greater number of opportunities, thereby performing well on CPU. However, while *img* reduced the loads by reusing computed values, it increased the length of the data dependency chain. This led to the *img* design having one extra state in its finite state machine created during scheduling due to data constraints, thus increasing its latency in the HLS setting.

This simple example shown above demonstrates that there are very subtle details and side effects that can have different impacts on CPU code and HLS designs. The impact of one optimization can be more pronounced in an HLS setting than in a CPU design. A typical CPU has many hardware features that enhance the performance of code that is being executed. For example, multiple levels of caches, out-of-order execution and load/store queues drastically reduce the cost of a single load. Branch prediction and speculative execution can hide the cost of evaluating a branch most of the times in case of loops. High-level synthesis is a different area in that way where each load corresponds to a read from a memory block, and each load costs the same number of cycles. Every branch instruction is dependent on another instruction that computes the exit condition, and branch

[1] We use the letters within brackets to refer to the respective optimizations in this section.

prediction mechanisms have to be specified in software manually by the designer if needed. Also, HLS can potentially exploit greater ILP limited only by the physical resources available on the target platform. On a typical processor, only the ILP available within the instruction window is exploited.

In this paper we perform an initial investigation into the impact of compiler transformations in a high-level synthesis setting (HLS). High-level synthesis is an automated design process that takes an un-timed, behavioral description of a circuit in a high-level language like C, and generates a register-transfer-level (RTL) net-list that implements the same behavior. The RTL generated by a HLS process is heavily influenced by the way the design is specified at the high level, making high-level optimizations very significant in the design flow. Works like [10] have tried solving similar problems in the HLS community in the past. We describe and use a set of simple, yet effective heuristics to quickly search the space of the described optimizations and study their effects on several benchmarks.

We also study the impact of classical IR-level optimizations on high-level synthesis. We evaluate several approaches, and suggest a new approach based on *lookahead* for optimizations. We also analyze two real-world benchmarks in a CPU-based and HLS-based setting and show how optimizations can have contrasting side effects. Our initial experiments show that latency improvements of more than 3X can be achieved by choosing the right order for an input behavioural description.

The rest of this paper is organized as follows. We provide some necessary background information regarding HLS and xPilot in Section 2. Our study on high-level optimizations is described in Section 3. We describe our methodology to search the space of IR-level optimizations in Section 4. We provide a detailed evaluation of our approaches in Section 5. We conclude with comments on future work in Section 6.

2 Background

High-level synthesis (HLS), or behavioral synthesis, is the process of automatically generating cycle-accurate RTL models from behavioral specifications. The behavioral specifications are typically in a high-level language, like C/C++/Matlab. The generated RTL models can then be accepted by the downstream RTL synthesis flow for implementation using ASICs or FPGAs. Compared to the traditional RTL-based design flow, potential advantages of HLS include better management of design complexity, code reuse and easy design-space exploration.

HLS has been an active research topic for more than 30 years. Early attempts to deploy HLS tools began when RTL-based design flows were well adopted. In 1995, Synopsys announced Behavioral Compiler, which accepts behavioral HDL code and connects to downstream flows. Since 2000, a new generation of HLS tools have been developed in both academia and industry. Unlike many predecessors, many of them use C-based languages to capture the design. This makes them more accessible to algorithm and system designers. It also enables hardware and software to be specified in the same language, facilitating software/hardware

co-design and co-verification. The use of C-based languages also makes it easy to leverage new techniques in software compilers for parallelization and optimization. As of 2012, notable commercial C-based tools include Cadence C-to-Silicon Compiler, Calypto Catapult C (formerly a product of Mentor Graphics), Synopsys Synphony C and Xilinx AutoESL (originating from the UCLA xPilot project [4]). More detailed surveys on the history and progress of HLS are available from sources such as [8] [5].

xPilot [4] is an academic HLS tool developed at UCLA. It takes as input a C function and generates an RTL Verilog module to implement the functionality. Compiler transformations are first performed on the source code using LLVM [14] to obtain an optimized IR, which can be translated to a control-data flow graph (CDFG). Scheduling is then performed on the CDFG to generate a finite-state machine with data path (FSMD) model, where each operation is assigned to a state in the FSM. Binding is then performed on the FSMD to allocate functional units, storage units and interconnects, and then the RTL net-list is decided.

For a given CDFG, the scheduler in xPilot tries to minimize worst-case latency by default, under the constraints of data dependency, control dependency, clock frequency, and resource limits [6]. The scheduler tries to insert clock boundaries on certain edges of the dependency graph, in order to guarantee that the delay and resource constraints are met. In a simplified model, operations in the same basic block are scheduled into consecutive control states; branches (including loops) are implemented as state transitions in the FSM. Thus, the resulting FSM is somewhat similar to the control-flow graph of the input function. If the control-flow graph of the input function is reducible, it is possible to estimate the worst-case latency of the module given the trip counts of loops.

3 Source-Level Optimizations

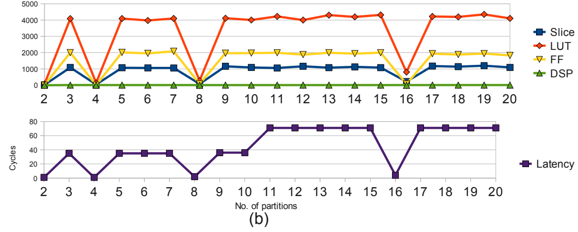
In this section, we describe our study of high-level optimization interactions. We consider three optimizations - *array partitioning*, *loop unrolling* and *loop pipelining*. We have chosen these optimizations as they are most commonly employed in standard high-level synthesis flow [1]. All the experiments in this section have been performed using AutoESL v1.0, 2011. [23]

3.1 Array Partitioning

Array structures in high-level design descriptions are implemented as memory blocks by default. However, mapping arrays to a single RAM resource can create resource constraints as each RAM block has only a few read and write ports. Mapping arrays to multiple RAM blocks can alleviate the resource constraint problem, provided the right number of banks are chosen. In this study, we concentrate only on cyclic distribution of array elements to different partitions. For example, consider a simple, contrived design as shown in Fig. 2(a). Fig. 2(b) shows the effect of partition factors on the latency and area. We make the observation that the best choices for the number of partitions are powers of 2. When

```
void testAP(int a[10], int i)
{
    a[i] = i;
}
```

(a)



(b)

Fig. 2. (a) Example design to study array partitioning. DSP: No. of DSP blocks, FF: No. of Flip-Flops, LUT: No. of Look-up tables.(b) Comparison of latency and area numbers for different partitions for design in (a).

an array is partitioned, additional code is inserted that performs *mod* operation on the index to select the right bank. Implementing *mod* on a power-of-2 number n just involves extracting the least significant $log_2(n)$ bits in the binary representation of n and truncating the rest, while for other numbers full 32-bit *mod* operation has to be realised in hardware. Such an operation is slow and occupies a lot of area.

3.2 Loop Unrolling

Loop unrolling is a popular optimization used to reduce loop overhead and increase ILP. It also exposes more opportunities to other optimizations like scalar replacement and dead code elimination. Consider two simple kernel loops shown

```
#define N 500
void daxpy(int a[N], int b[N], int k, int c)
{
   int i;
   L1:for(i=0;i<N;i++)
   {
      a[i] = b[i] * k + c;
   }
}
```

(a) No dependency

```
#define N 500
void prefix(int a[N+1], int b[N+1], int k, int c)
{
   int i;
   L1:for(i=1;i<N+1;i++) {
      a[i] = a[i-1] + a[i];
   }
}
```

(b) Dependence distance = 1

Fig. 3. Two simple kernels subject to loop unrolling

in Fig. 3. Fig. 4(a) and 4(b) show the latency and area numbers of the loops in Fig. 3. We make the following observations – (1) The best performing unroll factors in both the kernels considered are *2,4,5,8,10,16,20*. In general, the set of best unroll factors consists of both the factors of the loop trip-count as well as all powers of 2 lesser than the trip count. Unrolling a loop with a number that is not a factor of the trip-count adds the overhead of additional exit checks and branches. For non-power-of-2 unroll factors, the exit checks need a full *32-bit* comparators which are much slower, making them poor choices. Due to these reasons, the FSM created for this design during the scheduling phase is larger and complicated, thereby needing greater area to be implemented. (2)

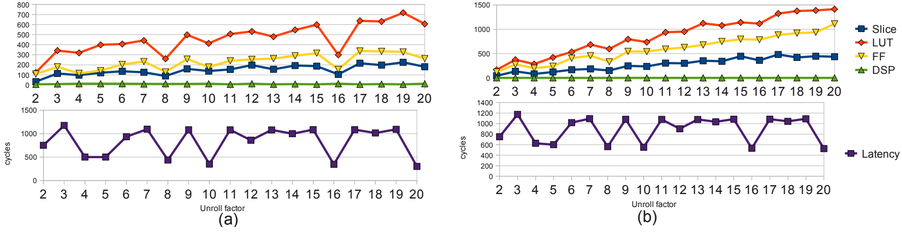


Fig. 4. Latency and area numbers for (a) Fig. 3(a) and (b) Fig. 3(b)

Area consumption increases linearly with unroll factor as it increases the size of one iteration and also the size of the FSM. (3) Unrolling loops with a carried dependency enables optimizations like scalar replacement and global value numbering. (4) Latency gain from unrolling quickly flattens out, while area does not. From the observations above, we form the following conclusions:

– The set of good unroll factors S for a loop L with a trip-count of n can be defined as follows:

$$S = \{f_i | mod(f_i, n) = 0\} \bigcup \{2^k | (k \in N) \wedge (2^k \leq n)\} \tag{1}$$

– Starting from the lowest unroll factor s_i in S, we iterate through the unroll factors and measure the relative drop in latency as well as relative increase in area. We continue our iterative search until we arrive at an unroll factor whose slope of area increase is greater than the slope of latency decrease, and return the previous best unroll factor at this stage. We use AutoESL's estimates to steer the algorithm as it is faster and accurate enough.

3.3 Loop Pipelining

Software pipelining is another popular loop transformation that also attempts to exploit ILP by re-ordering instructions across iterations and overlapping execution of consecutive iterations. Pipelining a loop with low initiation interval yields a high throughput. However, software pipelining can be constrained by the available memory bandwidth. Consider Fig. 5(a) for instance, where resource constraints is inhibiting pipelining. With appropriate array partitioning (Fig. 5(b)), software pipelining combined with loop unrolling proves to be a powerful combination.

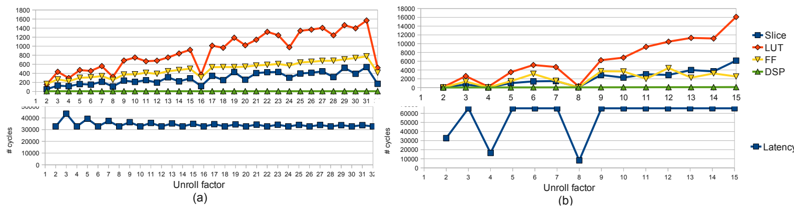


Fig. 5. Pipelining with unrolling loop in Fig. 3(a) for 65536 iterations in (a) without memory partitioning (b) with memory partitioning

3.4 Approach to Search Optimization Space

We use the algorithm described in section 3.2 to obtain the unroll factor u_i giving best performance to area. The loop is unrolled u_i number of times and then pipelined. If the II is constrained due to memory resources, the appropriate array is subjected to partitioning. The partition factor starts at 2 and is then doubled in subsequent iterations if the previous partition factor was insufficient to resolve the resource constraint. We discuss and evaluate our approach in section 5.

4 IR-Level Optimizations

In this section, we describe our study on the effects of phase-ordering of IR-level optimizations.

Optimizations Considered. By default, xPilot applies close to 250 transformations from a set of 55 unique optimizations. The optimization space is very discrete as can be seen in Fig. 6, which was obtained after evaluating 1000 random sequences of length 200 from the same optimization set. We first reduce the search domain in order to obtain greater insight. For this purpose, we randomly chose 100 sequences and examined the effect of each optimization in the sequence. Table 1 gives a brief description of all the short-listed optimizations. From here on in this paper, we restrict all our experiments to this restricted subset of optimizations.

Random Search. In our implementation of random search, we generate random sequences containing upto 25 optimizations each allowing repetitions. We generated and evaluated 5000 random sequences for each of the benchmarks considered.

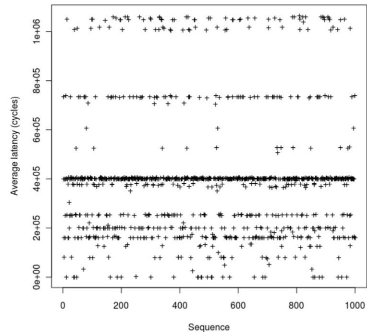


Fig. 6. Scatterplot of latencies for *matrixmul*

Genetic Algorithm. We implement a genetic algorithm to search the space of optimization sequences using latency as the minimization cost function. In our implementation, we chose to have a randomly generated initial population of 20 sequences, each of which can have upto 25 optimizations. We repeat the iterative search process for 500 generations. In each iteration, all the sequences in the population are evaluated and ranked. At the end of evaluation, sequences in the population undergo mutation and crossover. The best sequence is preserved as it is. Finally, 8 to 10 sequences are randomly chosen and mutated. Our implementation chooses sequences at the bottom with

a higher probability to be mutated by changing flags randomly. Finally, duplicate sequences are replaced with random sequences. The best solution found after 500 generations is reported.

***n*-Lookahead Scheme.** The n-lookahead scheme attempts to construct an optimization sequence by progressively deciding on the best subsequence of length n. It is based on an observation that the number of optimizations enabled or disabled by each optimization is relatively small. We are effectively *looking ahead* by n steps and choosing the subsequence that gives the best overall benefit at each step. Therefore, a 0-lookahead scheme is a greedy approach that chooses the best optimization successively and an N-lookahead scheme (where N is the length of the target sequence) is an exhaustive search. The parameter n provides a tradeoff between the amount of global information considered and number of comparisons. If we have to construct a sequence of length k with n levels of look ahead, and we have N number of unique optimizations, the number of combinations to be evaluated is $(\frac{k}{n}) * N^n$. Larger values of n increases number of sequences exponentially. In section 5, we evaluate the effectiveness of 0-lookahead and 1-lookahead schemes.

Table 1. Subset of optimizations and descriptions

Name	Description
adce (a)	Aggressive dead code elimination
bitwidthmin (b)	Bitwidth minimization
condprop (p)	Conditional propagation
constprop (k)	Constant propagation
dse (e)	Dead store elimination
gcse (c)	Global common subexpression elimination
gvn (n)	Global value Numbering
indvars (v)	Canonicalize induction variables
instcombine (i)	Combine redundant instructions
inst-simplify (t)	Operator strength reduction
loop-deletion (d)	Delete dead loops
loop-preproc (o)	Loop preprocess
loop-simplify (l)	Canonicalize natural loops
mem2reg (m)	Promote memory to register
ptr-legalization (r)	Convert pointers to array indices
simplifycfg (s)	Simplify the control-flow graph
xunroll (x)	Partially unroll loops

MSIR. We also evaluate an approach called Multi- Start Iterative Refinement (MSIR). In this approach, we generate N random sequences. Each sequence $(a_1, a_2 ... a_n)$ is subjected to an iterative refinement process as follows: Starting from the first pair (a_1, a_2), we generate two sequences starting with (a_1, a_2) and (a_2, a_1) choose the better sequence. We then move to the next pair in the chosen sequence (i.e., to the second position) and perform a similar evaluation. We continue iterating through pairs as long as we see improvement. The iterative search stops when no improvement is obtained upon one iteration through the entire sequence. The best sequence obtained from all the random sequences is returned. Section 5 evaluates and compares MSIR with the other described approaches.

5 Evaluation

5.1 Experiment Design Flow

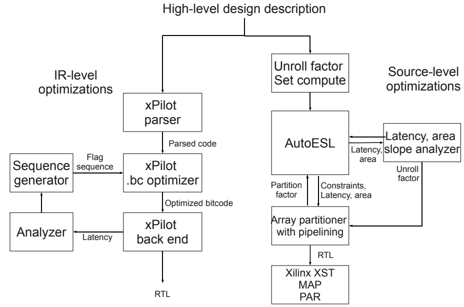


Fig. 7. Broad design flow used in all our experiments

Fig. 7 describes the architecture of our flow. Source-level transformations are studied using AutoESL. The final area numbers reported are from Xilinx's back-end tools. We use xPilot to study IR-level transformations and their impact on the latency of the RTL generated. As AutoESL is does not provide the user such fine-grained control to specify IR-level optimizations, AutoESL is not a suitable tool to study such lower level optimizations. We have modified xPilot to specify arbitrary optimization sequences. Also, as we do not study area utilization for IR-level transformations we do not go through the Xilinx back-end.

5.2 High-Level Optimizations

For purposes of evaluation, we use AutoESL v1.0, an industry-standard high-level synthesis tool. We obtain area numbers from the EDA tool-chain provided by Xilinx. The target platform we consider here is Xilinx Virtex-5.

Results. We tested our approach on five different kinds of kernels taken from the Open Accelerator repository [1] and MiBench [11]. We have hand-chosen different kernels in order to achieve a broader evaluation coverage. The benchmarks are described in Table 2. Overall, we achieve a mean reduction in latency

Table 2. Benchmarks for evaluation of high-level optimizations

Benchmark	Description
adpcm_decoder	Kernel function of the ADPCM decoding algorithm.
daxpy	kernel function performing the vector operation $A = Bk + c$
prefix	kernel function calculating prefix sum on a vector of integers
segmentation	Compute step in an image segmentation algorithm
smithwaterman	Smith-Waterman algorithm

of 50.42% over xPilot's default setting. Table 3 shows the factor obtained from our approach, number of partitions required, latency and area numbers for all benchmarks. Each benchmark is reported under three configurations: *Baseline* – where the benchmark was run without any high-level optimization; *Baseline + PP* – baseline with pipelining, where the main loop was pipelined with the

Table 3. Comparison between baseline and optimized benchmark versions against latency and area using ER

Benchmark	Unroll factor	No. of partitions	Numbers							
				Slice	LUT	FF	II	Depth	Latency	ER
adpcm_decoder	4	1	Baseline	200	588	217	-	-	2502	1
		1	Baseline + PP	224	741	234	2	5	1006	2.22
		1	U4 + PP	619	2009	471	8	11	1006	0.8035
daxpy	8	1	Baseline	21	80	62	-	-	1501	1
		1	Baseline + PP	26	92	75	1	3	504	2.405
		4	U8 + PP	89	324	315	1	3	67	5.286
prefix	8	1	Baseline	29	113	80	-	-	1501	1
		1	Baseline + PP	43	166	91	2	3	1003	1.009
		8	U8 + PP	109	307	375	2	4	130	3.072
segmentation	32	1	Baseline	31	110	65	-	-	8321	1
		1	Baseline + PP	43	153	88	1	2	4100	1.463
		16	U32 + PP	173	522	160	1	3	132	11.296
smithwaterman[2]	4	1	Baseline	26	102	46	-	-	52281	1
		1	Baseline + PP	19	73	46	2	3	11708	6.110
		1	U4 + PP	-	-	-	-	-	-	-

required number of array partitions; and $U(num) + PP$ – unroll by obtained unroll factor with pipelining along with required number of array partitions.

We define the *efficiency ratio* ER as the latency-area product, as follows:

$$ER = \frac{latency_b * area_b}{latency * area} \quad (2)$$

Here, $latency_b$ and $area_b$ are the latency and area numbers of the baseline respectively. We use the number of slices occupied as the representative for area of a design. We make the following observations:

- *adpcm_decoder* does not benefit from unrolling due to a scalar loop-carried dependency. Due to a scalar carried constraint, unrolling the loop does not increase ILP. Hence the best result for this benchmark is when unrolling is at its minimum i.e., the loop is completely rolled.
- *Segmentation* achieves a remarkable benefit with its configuration with an ER of around 11. As the core loop is data parallel, the only constraint to achieve minimal II would be array resources, and a partitioning factor of 16 resolves all resource constraints.
- smithwaterman benefits from pipelining with an ER of 6, and also shows impressive area usage. Using our heuristics, an unroll factor of 4 was found to give the best performance to area value. However, pipelining the unrolled loop resulted in an AutoESL crash due to an internal bug in the tool.

5.3 IR-Level Optimizations

In this section, we evaluate our approaches described in section 4. Table 4 lists and describes the set of benchmarks that we consider in our evaluation process. We have chosen a few different benchmarks in this section because xPilot is not as mature a tool as AutoESL and failed to synthesize some of the benchmarks.

Table 4. Benchmarks and description

Benchmark	Description
binarysearch	Iterative binary search
cftmdl	Kernel region in 1D FFT computation
chem	DSP algorithm in a chemical plant
dir	Direct implementation of 1D DCT
fft	Fast fourier tranform from MiBench [11]
honda	DSP filter application
jacobi	Jacobi method to solve linear equations
lee	Lee's algorithm for 1D DCT [15]
matrixmul	Tiled matrix multiplication
sha	SHA-1 encryption algorithm
smithwaterman	Smith-Waterman algorithm

Random Sampling vs. xPilot. Fig. 8 shows the comparison between the results of random search and the default optimization setting in xPilot. It can be seen that there are significant gains that can be achieved with an optimization order that is benchmark-specific. Overall, we achieve a mean reduction in latency of 50.42% over xPilot's default setting.

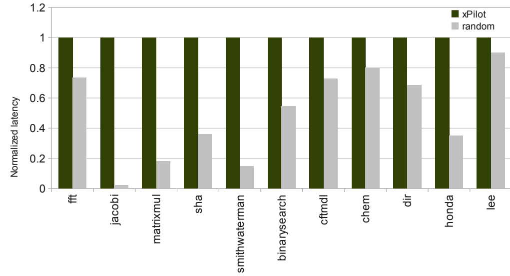


Fig. 8. Comparison of normalized latencies. *Default*: xPilot's default sequence.

Comparison of Approaches. We compare the performances of various approaches discussed in Section 4. We include only one small benchmark (*binarysearch*) for brevity as a representative example for all the other smaller benchmarks in all our further analyzes.

We can observe from Table 5 that random search and genetic algorithm match up to each other in most cases except *fft*. We can also observe that a similar trend exists between random search and 1-lookahead. We consider this a promising result, as we can achieve the same result as random search in lesser comparisons. We can observe that *mem2reg* is the sole critical optimization for *sha*. Also, we found that *jacobi* suffered with the default sequence due to a disabling interaction between *-scalarrepl* and *-gvn*.

Comparison with CPU Performance In order to compare the HLS setting with a CPU-based setting, we picked 200 of the randomly generated optimization sequences for two benchmarks, *sha* and *smithwaterman*. 200 executables were created, labeled with their sequence and simulated using Simics to get accurate cycle counts. Each sequence was given a *CPU rank* and an *xPilot* rank based on their execution time and latency respectively.

Fig. 9 shows the rank disparity between xPilot and CPU for the same optimization sequence. Consider a specific example. It is surprising to see that Sequence 1101 for benchmark *Sha* has an xPilot rank of 2 but a CPU rank of 175: `ldaomboptcxrp`

Our analysis shows that the HLS-specific *bitwidth* optimization was adding a lot of overhead instructions to obtain the operands of the appropriate width, thereby increasing the instruction count. Such side-effects do not exist in HLS because an operator of a specific bit-width can be realized in hardware. We

Table 5. Comparison of different approaches. The *Gen:* field shows the number of generations taken to converge. MSIR has been evaluated with N=10 random sequences. We generated 5000 sequences to evaluate random search.

Benchmark	Approach		Benchmark	Approach	
fft	Random	Latency: 2361 Sequence: srnaln	jacobi	Random	Latency: 492 Sequence: cbaemkxemsbsdtntsem
	GA Gen: 186	Latency: 1896 Sequence: kmnsosbcsenainr		GA Gen: 33	Latency: 492 Sequence: bmciensi
	0-lookAhead	Latency: 2359 Sequence: nktadnxpslbcoemns		0-lookAhead	Latency: 48358 Sequence: mxbtpsealodk
	1-lookAhead	Latency: 2339 Sequence: mnriksbtvbosknkacr		1-lookAhead	Latency: 492 Sequence: kmnsossbcsebnaiinr
	MSIR	Latency: 5359 Sequence: pmkasr		MSIR	Latency: 66116 Sequence: mxbeon
matrixmul	Random	Latency: 49 Sequence: aooeakbesmdsttvocaosa	sha	Random	Latency: 1442 Sequence: mxxasdotxrlxlbrmnt
	GA Gen: 28	Latency: 49 Sequence: mceosi		GA Gen: 1	Latency: 1442 Sequence: iiiiiiim
	0-lookAhead	Latency: 80119 Sequence: rnevxbntoldamssss		0-lookAhead	Latency: 1442 Sequence: mxnapseixrnacsmo
	1-lookAhead	Latency: 669 Sequence: rpnevxbpdeboomisiceb		1-lookAhead	Latency: 1442 Sequence: kmsipekrbrrpemoker
	MSIR	Latency: 368094 Sequence: kbrmae		MSIR	Latency: 1442 Sequence: cltpmx
smithwaterman	Random	Latency: 23 Sequence: kbvdsbbtmeosmn	binarysearch	Random	Latency: 12 Sequence: etaneb
	GA Gen: 51	Latency: 23 Sequence: inemsobs		GA Gen: 1	Latency: 12 Sequence: iiiiiiin
	0-lookAhead	Latency: 844 Sequence: kneimmissnirnxxxp		0-lookAhead	Latency: 12 Sequence: mtsnprciebvsdsak
	1-lookAhead	Latency: 23 Sequence: neamiaosaaaa		1-lookAhead	Latency: 12 Sequence: kmpkrsnciienaebmts
	MSIR	Latency: 2765 Sequence: vsamdn		MSIR	Latency: 12 Sequence: obrnxs

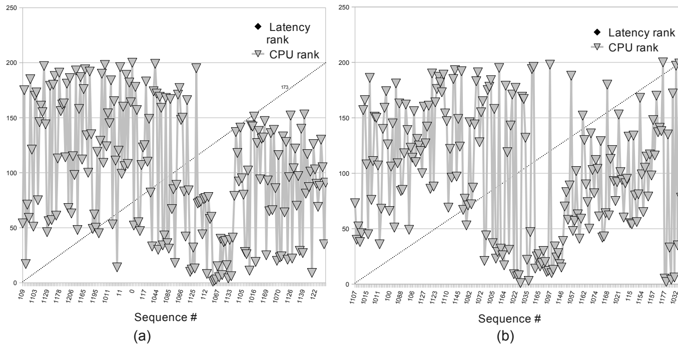


Fig. 9. Rank comparison for CPU vs xPilot. (a) Sha. (b) SmithWaterman.

generated another binary using the same sequence without the *-bitwidthmin* optimization, and observed that the CPU cycle count dropped from 70790 to 55981, which is very close to the lowest value of 48673. We also found that most of the lower-ranked sequences for CPUs had *-bitwidthmin* optimization included in them. Consider the opposite case in optimization sequence 118, which has a CPU rank of 9 and an xPilot rank of 191: `npxvervadkxnrnlx`

The disparity in ranks is caused due to an interesting interplay between *-gvn* and *-indvars* in the CPU-based and HLS-based settings. We performed additional experiments summarized in Table 6. We can see that the pair *-gvn*

Table 6. Comparison of CPU and HLS settings with optimization sequences involving *-gvn* and *-indvars*

Optimization sequence	xPilot latency	CPU cycle count
(none)	1844	54710
-indvars	1844	54709
-indvars -gvn	1444	54959
-gvn	1444	54958
-gvn -indvars	4024	53644

and *-indvars* affect the CPU and HLS setting in opposite ways. Intuitively *-gvn* decreases number of instructions and our simulation runs confirm that. While *-gvn* removes redundant code, it can have a potential side effect of introducing data dependency due to re-use. Also, if the data to be re-used is in memory, *-gvn* can slightly increase the number of loads in the program, as is the case in our example. The combined effect leads to an increased number of pipeline stalls which explains the increased cycle count. The *-indvars* pass increases code size and also tends to promote certain memory values to registers, thereby reducing the number of loads. Running an *-indvars* pass after *-gvn* pass effectively undoes the damage caused by *-gvn*. Hence we see that in the CPU setting, *-indvars* has a positive effect after *-gvn*.

In the HLS setting, however, there is no instruction execution pipeline. The design can be seen as a data and control-flow driven application where a finite set of instructions can be scheduled to run in every cycle. Hence, fewer instructions need fewer cycles to run. This explains the positive effect of *-gvn* on the xPilot latency. Running an *-indvars* pass after *-gvn* introduces many additional instructions and dependencies, increasing the number of FSM states and latency. We re-ran xPilot using sequence 118 without the *-indvars* option and observed that the latency dropped from 4024 to 1444.

The above experiments show convincingly that applying good transformations for a CPU may not lead to good HLS results, and HLS-specific code optimization sequences and transformations are needed.

6 Conclusions

Given the rise in popularity in high-level synthesis as a popular design choice in the system design community, we believe that having a sound compilation technology in high-level synthesis is very essential. In this paper, we have presented a first study on the impact of compiler optimization phase ordering on design space exploration and the quality of the generated RTLs. We have presented studies on three important high-level optimizations - loop unrolling, software pipelining and array partitioning. We have described simple heuristics to quickly eliminate bad choices early. We have also presented a detailed study of the IR-level optimization phase ordering on high-level synthesis where a variety of techniques were discussed and evaluated. We reported a mean reduction in

latency of 50.42% against xPilot's default setting. We compare our study to a CPU-based setting and provide several insights into the subtle variations that causes pairs of optimizations to have different effects.

We believe that our work opens up many interesting future directions. An interesting direction for future work would be to consider more high-level optimizations. With the vast amount of data that we have collected, the idea of building a predictive model using program featurs seems attractive as well. With our promising initial results, we believe that research in this direction would benefit the high-level synthesis community.

Acknowledgements. The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity, and also the support from Altera Corporation.

References

1. Open Source Accelerator Store, http://cadlab.cs.ucla.edu/accelerator_store.html
2. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M.F.P., Thomson, J., Toussaint, M., Williams, C.K.I.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2006, pp. 295–305. IEEE Computer Society, Washington, DC (2006)
3. Chabbi, M.M., Mellor-Crummey, J.M., Cooper, K.D.: Efficiently exploring compiler optimization sequences with pairwise pruning. In: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT 2011, pp. 34–45. ACM, New York (2011)
4. Cong, J., Fan, Y., Han, G., Jiang, W., Zhang, Z.: Platform-based behavior-level and system-level synthesis. In: Proc. IEEE Int. SOC Conf., pp. 199–202 (2006)
5. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for FPGAs: From prototyping to deployment. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 30(4), 473–491 (2011)
6. Cong, J., Zhang, Z.: An efficient and versatile scheduling algorithm based on SDC formulation. In: Proc. Design Automation Conf., pp. 433–438 (2006)
7. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems, LCTES '99, pp. 1–9. ACM, New York (1999)
8. Coussy, P., Morawiec, A.: High-Level Synthesis: From Algorithm to Digital Circuit. Springer (2008)
9. Epshteyn, A., Garzarán, M.J., DeJong, G., Padua, D.A., Ren, G., Li, X., Yotov, K., Pingali, K.K.: Analytic Models and Empirical Search: A Hybrid Approach to Code Optimization. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 259–273. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/978-3-540-69330-7_18
10. Gupta, S., Gupta, R.K., Dutt, N.D., Nicolau, A.: Coordinated parallelizing compiler optimizations and high-level synthesis. ACM Trans. Design Autom. Electr. Syst. 9(4), 441–470 (2004)

11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Workload Characterization, WWC-4, 2001 IEEE International Workshop, pp. 3–14. IEEE Computer Society, Washington, DC (2001)
12. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P.: Combined selection of tile sizes and unroll factors using iterative compilation. In: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, PACT 2000, p. 237. IEEE Computer Society, Washington, DC (2000)
13. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast searches for effective optimization phase sequences. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI 2004, pp. 171–182. ACM, New York (2004)
14. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. Int. Symp. on Code Generation and Optimization, p. 75 (2004)
15. Lee, B.: A new algorithm to compute the discrete cosine transform. IEEE Trans. Acoustics, Speech and Signal Processing (6), 1243–1245 (1984)
16. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2006, pp. 319–332. IEEE Computer Society, Washington, DC (2006)
17. Pollock, L.L.: An approach to incremental compilation of optimized code. PhD thesis, Pittsburgh, PA, USA, UMI order no. GAX86-20225 (1986)
18. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2005, pp. 123–134. IEEE Computer Society, Washington, DC (2005)
19. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.-M.: Meta optimization: improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 77–90. ACM, New York (2003)
20. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 264–276. ACM, New York (2009)
21. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2003, pp. 204–215. IEEE Computer Society, Washington, DC (2003)
22. Whitfield, D., Soffa, M.L.: An approach to ordering optimizing transformations. In: Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, PPOPP 1990, pp. 137–146. ACM, New York (1990)
23. Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., Cong, J.: AutoPilot: A Platform-Based ESL Synthesis System, pp. 99–112 (2008)
24. Zhao, M., Childers, B., Soffa, M.L.: Predicting the impact of optimizations for embedded systems. In: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, LCTES 2003, pp. 1–11. ACM, New York (2003)
25. Zhao, M., Childers, B.R., Soffa, M.L.: A Framework for Exploring Optimization Properties. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 32–47. Springer, Heidelberg (2009)