Int J Parallel Prog (2016) 44:337–380
DOI 10.1007/s10766-015-0380-7

Memory Partitioning in the Limit

Emre Kültürsay[1] · Kemal Ebcioğlu[2] · Gürhan Küçük[3] · Mahmut T. Kandemir[1]

Received: 4 July 2013 / Accepted: 18 September 2015 / Published online: 26 October 2015
© Springer Science+Business Media New York 2015

Abstract The key difficulties in designing memory hierarchies for future computing systems with extreme scale parallelism include (1) overcoming the design complexity of system-wide memory coherence, (2) achieving low power, and (3) achieving fast access times within such a memory hierarchy. Towards addressing these difficulties, in this paper we propose an automatic memory partitioning method to generate a customized, application-specific, energy-efficient, low latency memory hierarchy, tailored to particular application programs. Given a software program to accelerate, our method automatically partitions the memory of the original program, creates a new customized application-specific multi-level memory hierarchy for the program, and modifies the original program to use the new memory hierarchy. This new memory hierarchy and modified program are then used as the basis to create a customized, application-specific, highly parallel hardware accelerator, which is functionally equivalent to the original, unmodified program. Using dependence analysis and fine grain valid/dirty bits, the memories in the generated hierarchy can operate in parallel with-

✉ Emre Kültürsay
euk139@cse.psu.edu

Kemal Ebcioğlu
kemal.ebcioglu@global-supercomputing.com

Gürhan Küçük
gkucuk@cse.yeditepe.edu.tr

Mahmut T. Kandemir
kandemir@cse.psu.edu

1 Pennsylvania State University, 111N IST Building, University Park, PA 16802, USA

2 Global Supercomputing Corporation, P.O. Box 603, Yorktown Heights, NY 10598, USA

3 Faculty of Engineering and Architecture, Yeditepe University, Office A-407, Kayisdagi, 34755 Istanbul, Turkey

out the need for maintaining coherence and can be independently initialized/flushed from/to their parent memories in the hierarchy, enabling a scalable memory design. The generated memories are fully compatible with the memory addressing in the original software program; this compatibility feature enables the translation of general software applications to application-specific accelerators. We also provide a compiler analysis method to perform accurate dependence analysis for memory partitioning based on symbolic execution, and a profiler-based futuristic limit study to identify the maximum gains that can be achieved by memory partitioning.

Keywords Memory partitioning · Parallel processing · Application-specific hardware accelerators · Exascale computing · Supercomputers

1 Introduction

As we move towards the era of exascale computing, high performance computing systems will exhibit an unprecedented degree of parallelism. As of today, achieving exascale performance is a hard problem, whose solution will likely include application-specific hardware, at least to reduce the total power consumption within the system. One example of an exascale system is therefore an application-specific hardware accelerator that spans a large number of racks and modules within these racks, each module housing custom hardware chips (ASIC or FPGA) with many small processing elements in them operating together to improve the performance of a particular application. In such a system, there will be thousands of memory accesses issued simultaneously; thus, the memory system must be very scalable to serve this many requests with low latency and low power. Yet, preferably, the memory system will need to comply with the semantics of a single unified, coherent memory (as if all processing elements were jointly working on one single sequential program) for improved programmer productivity.

Our research is directed towards making a breakthrough in the memory designs of future high performance systems. We believe that, by using automatic memory partitioning, we can solve the memory scalability problem by creating custom, application-specific memories, thus achieving less design complexity, lower power, and faster access times through hardware specialization. Memory partitioning [24,47] is a technique to decompose the memory address space of a software program into independent subspaces, such that each subspace is implemented by an independent memory partition that is accessed by only a subset of the processing elements/memory request sources in the system. For the purpose of building large parallel hardware systems, the most important benefit of using partitioned memory over distributed shared memory is the simplification in coherence hardware: the customized memory system created for the program is exactly equivalent to the original program memory, yet the generated memories do not need to be kept coherent. As a result, this partitioned memory system can easily scale to a large number of memories that can operate in parallel to satisfactorily serve all the simultaneously issued requests.

1.1 Contributions

We developed a compiler/profiler framework to analyze programs, create a specialized partitioned memory for these programs, and quantify the benefits of using this partitioned memory. Specifically, we make the following contributions with this work:

- Our first contribution is a method to partition memory. Our method differs from existing methods in two aspects. First, the memories generated by our method are 100 % compatible with the corresponding addresses used by the original software program, thus widening the applicability of memory partitioning to arbitrary, general, sequential code (including assembly code). Address compatibility is the key to enabling full functional equivalence between an arbitrary software program fragment and the hardware accelerator it is compiled into. Second, our method hierarchically partitions the memory space of the target program. This results in a larger number of smaller memories that are active during the execution of inner program scopes, which can further improve performance and energy efficiency of the memory system.
- Our second contribution is an enhancement to symbolic execution, a technique which can lead to more aggressive and more accurate dependence analysis with optimizing compilers [21]. We extend symbolic execution to handle languages with aliasing and propose heuristics to address its computational complexity. We use this symbolic execution method to perform dependence analysis, the results of which are used for memory partitioning.
- Our third contribution is a profiler-based method to identify the maximum degree of memory partitioning that can be achieved for a given application.
- Finally, our fourth contribution is a detailed evaluation of various SPEC benchmarks [27,45] from the memory partitioning perspective. We provide compiler-based and profiler-based memory partitioning results. The profiler-based results denote the maximum degree of memory partitioning (i.e., *the limit*) that can be achieved using speculative memory partitioning schemes. By comparing the compiler and profiler-based results, we also identify input and compiler-related limits to memory partitioning.

1.2 Organization of this Article

The remainder of this article is organized as follows. In Sect. 2, we first describe our target architecture, namely, application-specific hardware accelerators, and then give an overview of our compiler/profiler toolchain. Sect. 3 summarizes the related work in the area and underlines the novel parts of this study. Sect. 4 formulates the memory partitioning problem and provides our approach to it. As memory partitioning relies on accurate disambiguation of memory references in the target program, we provide a detailed description of our dependence analysis methods in Sect. 5. This includes a static analysis method performed by our compiler as well as a dynamic analysis method performed by our profiler. We provide the details of our experimental evaluation in Sect. 6 and conclude with Sect. 7.

2 Overview of the Proposed System

2.1 Application-Specific Hardware Accelerators and the Role of Automatic Memory Partitioning

Application-specific hardware accelerators—in the form of ASICs or FPGAs—are typically generated by translating some part of a target software program (such as one or more hot loops or procedures) into custom hardware, accompanied with a method to transfer execution control and data across the software program and the generated hardware accelerator. Execution of this custom hardware is equivalent to the original software program fragment it is extracted from, but takes a shorter time, thereby improving the performance of the original application. Execution of the application typically starts in software running on the host, transferring control (and necessary input data) from the host to the accelerator at the entry of the code fragment mapped to custom hardware, and returning back to the host software (and copying necessary output data) at the end of hardware execution.

Custom hardware-based accelerators include various features that enable them to outperform general purpose processors. While prior work does not describe any general method to convert arbitrary sequential program code into a highly parallel custom hardware-based accelerator, at least on scientific array codes custom hardware-based accelerators work well and can exploit lots of parallelism, both at a coarse grain thread level and at a fine grain instruction level [10,23]. Compared to general purpose processors, such custom hardware-based accelerators also do not suffer from various intrinsic overheads, such as the overheads associated with instruction fetch/decode/issue and overheads due to synchronization.

In this work, we start with a sequential program code as input and realize parallelism in the application-specific hardware corresponding to this code, by using hierarchical software pipelining and resource duplication. Using software pipelining [34], each (outer or inner) loop within the loop hierarchy of the target program is mapped into a pipelined state machine. Applying software pipelining hierarchically, an outer loop can dispatch the execution of an inner loop to an inner loop state machine by simply treating the inner loop as a multi-cycle pipelined operation. Duplicating the loop state machine resources enables parallel execution of multiple inner loop instances. When an outer loop state machine reaches the start point of an inner loop, it dispatches an inner loop execution operation to one of the available inner loop state machines over a dedicated scalable network, following a “spawn-and-forget” thread model. The inner loop thread will eventually run when an inner loop state machine becomes free. To maintain sequential semantics, synchronization across threads, if necessary, is performed by employing specialized, application-specific synchronization networks in the custom hardware. This nested parallelization strategy enables a large number of state machines to execute simultaneously in the application-specific hardware system. As a result, a large number of hardware units can issue requests to the memory system, resulting in a large contention in the memory system. Possibly recursive subroutine calls can be treated in the same manner as an inner loop invocation.

Another salient feature of custom hardware accelerators is that they typically use a specialized, partitioned memory architecture [15,47]. Partitioned memory is synthe-

sized at compile time together with the rest of the hardware accelerator. In comparison to the single, unified memory view in general purpose systems, custom hardware has the flexibility to use multiple smaller memories designed specifically for the target program. These small memories, when combined, reproduce the information content in the original software program, which means that partitioning does not alter the semantics of the application. The advantage of using partitioned memories is that these memories are accessed independently without the need for any coherence hardware across different memories. As a result, smaller networks can be used to connect the compute units to the memories, which can in turn enable higher throughput at a lower complexity. Individual memory partitions are smaller, preferably small enough to be realized using on-chip resources. This means that they are faster and more energy efficient, further enhancing the performance of the accelerator.

In this work, we use a hierarchical memory partitioning method that is able to automatically create a multi-level memory hierarchy to realize a scalable memory system. Figure 1 shows the single, unified memory of a software code fragment partitioned into a hierarchy of memories in its application-specific hardware equivalent. The software code fragment memory (`MEM0` acts as the accelerator's last level cache for the software application memory), in this example, is partitioned at a procedure level (level 1, yielding children `MEM1` and `MEM2` of `MEM0`) and at an outer loop level (level 2, yielding children `MEM3` and `MEM4` of `MEM1`) to obtain two new memory levels on the accelerator side. The memories at the same level (e.g., `MEM1` and `MEM2` (level 1) or `MEM3` and `MEM4` (level 2)) do not have any coherence hardware across them, as the compiler proved that there is no data dependence across the processing elements/memory request sources (i.e., software code fragment instructions) connected to these memories over the target program region. The elimination of coherence hardware simplifies the hardware design, and realizes a more scalable memory system. Each memory, in turn, can be further partitioned into smaller child memories that once again do not have coherence hardware across them. Child memories are connected to their parent memories using specialized hardware that is used for initializing and flushing of the memories at the entry and exit points of the target program regions, when needed. Notice that as we duplicate compute resources in order to exploit parallelism, we also duplicate the memories in the system, adding a third dimension (into the page) to the memory hierarchy design shown in this figure. Thus, the replicated `j` loop state machines in Fig. 1 each have their own private `MEM3` and `MEM4` memories. Each node in the memory hierarchy can be implemented using SRAMs or flip flops when the size of the data structures contained in the memory node is known at compile time and is small, or by using a cache, otherwise.

It is typical for the existing synthesis tools to generate a system with distributed memory, where the host and the accelerator operate at different address spaces. In this case, a data structure has two distinct addresses on the two domains and copying across address spaces requires an address translation. While this translation is straightforward for simple data structures (e.g., an array only needs its base address to be translated), complex data structures cannot be handled so easily. When a host address is stored in the data structure as data, such as linked list node keeping a pointer to the next node, address translation is needed not only for variable addresses, but also addresses stored as data in pointers. This complicates the host-accelerator memory coherence

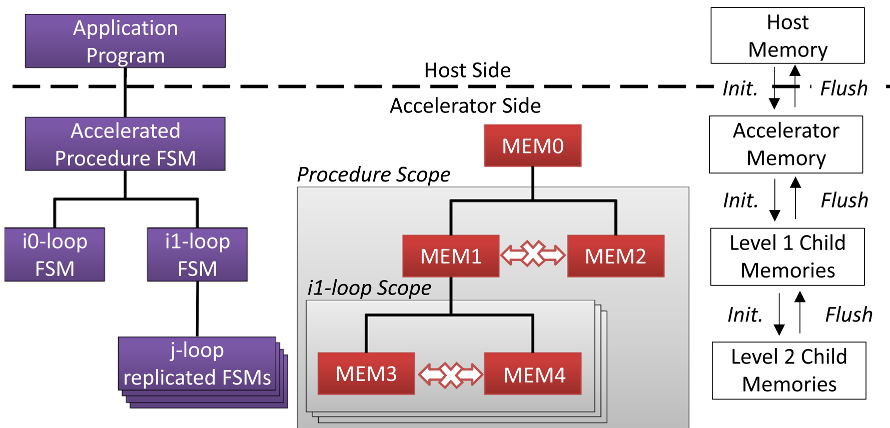

Fig. 1 An example memory hierarchy created by our memory partitioning method. The scope (loop) hierarchy of the target program fragment is used to hierarchically partition the memory in the original application. The unified memory of the application `MEM0` is first partitioned into two memories `MEM1` and `MEM2` at the entire accelerated procedure level. Then, `MEM1` is further partitioned into two memories `MEM3` and `MEM4` at the `i1` loop level. Different instances of the `j` loop enclosed by the `i1` loop can execute in parallel, and as a result, these second level partitioned private memories of the `j` loop are duplicated. Memories residing at the same level of the hierarchy need not have any coherence hardware across them. The source code corresponding to this example will later be analyzed in Fig. 4

problem, and most accelerator systems prohibit the use of complex data structures. The method we propose in this work can partition the memory of applications with data structures of arbitrary complexity using distributed shared memory, which is based on full address compatibility across the host and the accelerator memories. As a result, our memory partitioning method can be applied to arbitrary sequential program code fragments, even assembly language code fragments, duplicating the function of all the original software loads/stores in the generated hardware accelerator; it is not limited to hot loops or array calculations.

2.2 Infrastructure

In order to perform and evaluate memory partitioning, we developed a new program analysis and optimization framework. This framework consists of a compiler and a profiler that are both developed from scratch. In this section, we give an overview of our compiler and our profiler, and how we use them together in our setup. Details of the algorithms used in these tools are provided in later sections.

Compiler There are four major steps carried out by our compiler. First, it parses the input program to construct an intermediate representation for applying analyses and optimizations, and performs basic optimizations on this intermediate code. While the input programming language we used in this study is x86 assembly language[1]

[1] Due to past progress in the binary translation and optimization field, the binary instructions of an ISA are by now a *de facto* compiler intermediate language [17,18,43] which allows access to a wide range of applications.

generated by the gcc 4.5.4 compiler with the "-gstabs+" option ("-gstabs+" adds a degree of source-level debug information to the assembly code), any input language can be used without loss of generality. The compiler then performs a symbolic execution of this intermediate representation of the program to disambiguate memory references in it. The third step is to partition the memory instructions in the input program. Without memory partitioning, all instructions in the input program access a single memory that comprises the entire address space of the program (e.g., the entire 32-bit or 64-bit address space). Memory partitioning creates a set of memories with smaller address spaces, organized as a hierarchy. Each memory operation in the input program is mapped to one memory in the extracted hierarchy. This hierarchy of memories is scalable: the hierarchy can include a large number of memories, as all memories are independent and there is no need for coherence hardware across them. The fourth and final step in our compiler is code generation. In this step, one can generate an application-specific supercomputer system at the Verilog RTL design level, with the memory hierarchy and the modified program created in the memory partitioning step, along with a modified software executable which communicates with the application-specific supercomputer when an accelerated code fragment is entered. The flat component netlist for the entire application-specific supercomputer is then partitioned into multiple chips interconnected by a scalable network, and a *union chip* is created which can act as any one of the partitions, so that only one ASIC chip needs to be released to build the system, thus reducing the non-recurring engineering costs. FPGA implementations of the supercomputer are also possible. Since the primary focus of this study is not to evaluate application-specific supercomputers, but to perform a futuristic limit study on memory partitioning, we skip the hardware construction step in the scope of this paper: we only emit the results of memory partitioning.

Profiler There are two main tasks our profiler performs. First, it complements the compiler in dependence analysis. Note that any compiler-based dependence analysis has two weaknesses: (1) practical compiler analyses are limited by compilation time constraints, and even if compilation time were unlimited, as a result of the inherent unsolvable nature of the memory disambiguation problem, there shall always be some dependences that cannot be proved to exist or not to exist[2], (2) there can be some runtime information needed to identify that a dependence is not observed in practice for

[2] A compiler algorithm answering memory disambiguation questions without any mistakes is impossible. Assume by way of contradiction that there were such an algorithm f such that $f(p, I_1, I_2)$ precisely determines whether the following statement is valid "there exists an execution/initial state of program p where an instance of I_1 executes and an instance of I_2 executes and these instances refer to the same memory location". Then, by Kleene's (second) recursion theorem [40], a "contrarian" program can be constructed which runs the algorithm f on itself, and its memory instructions I_1 and I_2, before ever executing I_1 and I_2, and then, after getting f's answer, executes its remaining part: (1) making I_1, I_2 independent regardless of the initial state, if the algorithm f responds with YES (I_1, I_2 are dependent), and (2) making I_1, I_2 dependent on at least one initial state, if the algorithm f responds with NO (I_1, I_2 are independent). Contradiction. Therefore, no such algorithm f exists. Therefore, even though many current compiler algorithms are able to avoid false negative mistakes (i.e., mistakes which say there is no dependence when in fact there is), any disambiguation algorithm attempting to approximate f must on some inputs either not finish, or must make conservative (false positive) mistakes, and say I_1, I_2 are dependent when they are not. But this negative result has not discouraged compiler researchers, since approximate memory disambiguation algorithms have done well enough.

typical input values. Therefore, we use our profiler to dynamically analyze the memory locations accessed by each memory instruction in the input program and use this information to identify all real dependences across memory instructions. The second task of the profiler is to evaluate the impact of memory partitioning on the performance and the energy consumption of the memory system. The profiler uses the memory partition table output of the memory partitioning step of the compiler and emulates the execution of the program on a hypothetical system with unlimited resources and partitioned memory. For this purpose, the profiler instruments the program code such that (1) each memory instruction is treated as if it accesses the memory given by the partition table, and (2) the access statistics for each memory are recorded. These statistics are then used to calculate the performance and energy consumption of the memory system using physical metrics.

The profiler is implemented by adding instrumentation code to the original code and keeping both the original version of the code and the profiled version of the code ready for execution. To improve the speed of the profiling, we activate profiling only during periodic samples of execution, instead of profiling the entire execution of the application from the start to the end. To implement the sampled execution, we create *landing locations* at certain points in the original and its corresponding profiled code. When a sampling period is over, a control code checks if the execution is within the profiled or the original code. If the execution is in the profiling mode, we suspend the profiling, and we jump back to the corresponding landing location in the original version of the code and continue execution until the next sampling period is reached. This method is much faster but, of course, less accurate than the well-known full profiling method. The sampling period might be arranged to find satisfactory points in both performance and accuracy curves of profiling. For instance, with this approach, we successfully profiled the *mcf* benchmark in less than two minutes on a particular input, whereas the full profiling of the same benchmark required more than two hours of execution on the same input. Alternatively, to achieve better profiling performance, there is also a possibility to build custom processors on FPGA with efficient dependence profiling hardware.

Compiler/Profiler Toolchain Viewed at a high level—our compiler/profiler toolchain operates in three stages (as shown in Fig. 2):

- *Profiling-1* Instrument the application and profile it to record all data dependences that occur at runtime across all pairs of memory instructions. This stage is optional and is used only for profiler-based optimistic memory separation.
- *Compilation* Perform static program analysis to identify data dependences across all pairs of memory instructions. Compare the static dependence analysis analysis results with the dynamic dependence analysis results from *Profile-1* (if available) (1) to verify the correctness of compiler dependence analysis tests, and (2) to enable profile-guided speculative memory partitioning. Partition memory using dependence analysis results and generate a memory partition table.
- *Profiling-2* Instrument the program using the memory partition table, augmenting each memory instruction with code that updates the access statistics of the corresponding memory in the memory partition table. Emulate the partitioned memory

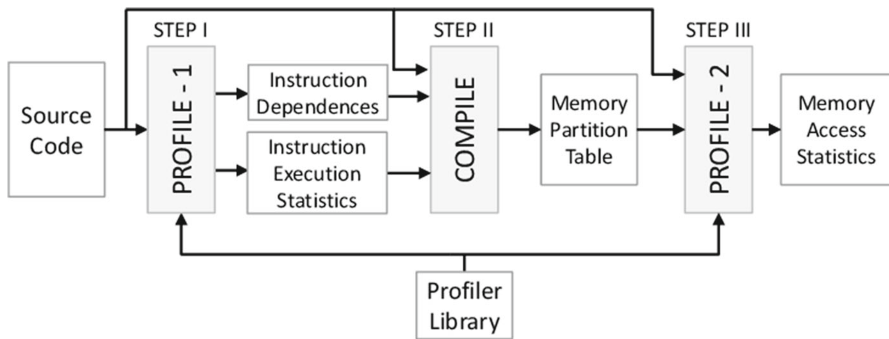

Fig. 2 High level view of our compiler/profiler infrastructure for analyzing programs and performing memory partitioning

system and obtain information about the usage patterns of the memories extracted by the compiler.

3 Related Work

In this section, we provide a summary of the prior work in the three areas where we make a contribution, and explain the features that distinguish our work from the prior work.

3.1 Compiler-Guided Memory Partitioning

Memory Partitioning for General Purpose Systems. Memory partitioning methods have been used in compilers for various purposes. A significant body of work has been directed towards identifying the optimum data layout on distributed systems, such that the communication across nodes due to remote memory accesses is minimized. By changing the original array layouts and distributing the parts of the arrays to the compute nodes in a locality-aware manner, the compiler can generate code that makes fewer remote memory accesses [1,2], and therefore, has better memory performance.

Another use of memory partitioning is to improve the performance of software managed, on-chip scratch pad memories [3,4]. In this case, the compiler applies memory partitioning over a program region to use different locations in the scratch pad memory for different variables throughout that region. The compiler also generates explicit instructions to load data from the main memory to the scratch pad before the first access to the target variables, and then writes any dirty result back to main memory after the last access is performed.

Partitioned Global Address Space (PGAS) languages [48] shift the memory partitioning task to the programmer. In PGAS languages [11,19,37,49], the programmer uses language extensions to provide a layout for data that will be shared by multiple compute units and the compiler generates the communication code for the given data layout. Although using PGAS languages for high performance computing is not as dif-

ficult as using Message Passing Interface (MPI), it still requires significant effort from the programmer to manually identify shared/private data and provide a data layout.

Notice that in a general purpose parallel processing system with a coherent memory hierarchy, even if software optimizations are present that reduce expensive communication, just to accommodate the possibility that a hardware component (memory, cache, or processing unit) A may communicate with another hardware component B, the network connecting A and B (e.g., multiplexers, wires, network protocol hardware) must be built and its costs must be incurred in terms of design complexity, added propagation delay, area, and power consumption. General purpose system-wide memory coherence hardware for large scale parallel systems is one of the most complex parallel hardware designs [26]. By contrast, special purpose application-specific hardware has the advantage that component A can be proved not to communicate with component B at hardware design creation time, and therefore, the design complexity, added propagation delay, area, and power consumption associated with this communication can be completely avoided. Moreover, specialized low-power hardware, as opposed to general purpose hardware, can be used to individually implement the components A and B. In contrast to the efforts mentioned above, which aim to improve memory performance of general purpose architectures, our goal in this work is to generate an application-specific memory system.[3]

Memory Partitioning for Application-Specific Hardware. The efforts that use memory partitioning in designing application-specific hardware share a common goal: by specializing the memory system based on the target application, one can improve performance while providing better energy efficiency. Given a processor with FPGA capability and with multiple local memories of different types, Gokhale and Stone [24] provide an algorithm that automatically allocates arrays in a program to the memories with the goal of reducing the execution time of software pipelined loops. Weinhardt and Luk [47] present a RAM inference and an array allocation method that generates on-chip RAM banks for arrays, which reduce the number memory accesses in FPGA coprocessors, also within pipelined inner loops. Benini et al. [7,8] synthesize custom multi-banked on-chip SRAM for SoCs such that memory accesses target smaller memories that consume less energy. It is important to note that they use memory partitioning in the context of a general purpose processor, and therefore observe energy benefits only due to reduced memory capacity. Since their memory interface is a general purpose bus, the memories they generate are still accessed using the same bus, whereas applying memory partitioning on custom hardware also enables trans-

[3] On scientific applications, e.g., loop nests containing arrays with affine subscripts, existing compiler transformations for parallelization and data localization on general purpose parallel processors could of course be used as a preprocessing step before applying our memory partitioning techniques for creating application-specific supercomputer hardware. For example, applying our technique on the loop `for(int i=0; i<nProc; ++i) func(i, data[i]);` created by a typical sequential to parallel code transformation [1] for scientific code (parallelized SPMD code converted back to a sequential representation) will lead to loop `i` iterations being dispatched through our hierarchical software pipelining on an array of identical application-specific hardware units implementing `func`, each with its private child memory hierarchy representing a partition of the `data` array, without using any coherence hardware across child memory hierarchies, and without using power-hungry general purpose processors to implement `func`. Our techniques, however, are not limited to scientific applications.

lation of such a shared medium into low-overhead point-to-point links. So et al. [44] propose a method to generate application-specific bank-interleaved data layouts for arrays within an FPGA environment, according to access patterns in the code. Their approach operates in conjunction with loop nest transformations that are commonly applied to array based computations and provide higher performance by increasing memory level parallelism. Baradaran and Diniz [6], Cong et al. [15] present efforts that combine scheduling techniques with memory bank-interleaved array layout to improve performance.

In comparison to these prior studies, our focus is on performing a limit study, where we make no restriction on how instructions are scheduled or how many requests can be simultaneously served by each memory.[4] Instead, our primary focus is to identify the maximum performance/energy benefits that can be achieved by using the largest number of memories and the smallest capacity memories that do not have coherence hardware across them. Our method also differs from all prior work with its capability to satisfy address compatibility with the original software program, its capability to generate a multi-level memory hierarchy, and its ability to accept general code (including assembly code) with arbitrary pointer memory accesses.

Software application programs can be very large, and not all of a software application can be converted to accelerator hardware: some parts will remain as software. Moreover, acceleration techniques confined to certain types of code, such as scientific loop nests containing arrays with affine subscripts, will likely not be effective in general, due to Amdahl's law. If a compiler is unable to accept general software, engineering productivity will be reduced due to the extra "porting" effort to convert the software part to be accelerated into a form required by the compiler, and to ensure that the rest of the software remains correct after the changes. Our compiler's ability to accept general code and its ability to achieve full functional equivalence between the accelerator hardware and the original software code fragment, including memory layout, address these issues not addressed by prior work.

3.2 Symbolic Execution in Compilers

Symbolic execution [30] has been used widely in various contexts [13,14,16,20,36], yet its use in optimizing compilers has been limited [21]. We identified two main deficiencies in its use in the prior work and devised solutions.

First, the only use of symbolic execution in an optimizing compiler has been with a compiler that uses Fortran as the input language. However, typical Fortran programs do not have any aliasing, except equivalence statements explicitly provided by the programmer. Large programs written using languages such as C/C++ typically make heavy use of aliasing that is not explicit. Therefore, we extended symbolic execution to handle arbitrary use of pointers. Since our input language is x86 assembly code, where even simple loop indices/induction variables may be allocated in memory, we built our general symbolic execution framework on program states comprising ("address"

[4] While the present work is a limit study, it is instructive to consider its practical implementation aspects. These aspects, including the implementation of parallel simultaneous accesses, will be addressed in Sect. 4.4.

symbolic expression, "contents of address" symbolic expression) pairs (as opposed to traditional [21] ("variable", "contents of variable" symbolic expression) pairs). Our symbolic execution framework therefore allows any program variable (including induction variables) to be accessed with arbitrary levels of indirection, and therefore is able to represent and analyze general sequential code including pointers (see Sect. 5.1).

Our second enhancement is related to the execution time of symbolic execution. Prior works have indicated the difficulty of employing symbolic execution in practical compilers due to its complexity [39]. To overcome this limitation, we use our own symbolic execution framework which is engineered for efficiency (we do not rely on any external symbolic math package, unlike [21]), and we use two heuristics that automatically trade off symbolic execution accuracy with performance. Our first heuristic handles cases where, due to the symbolic execution of a sequence of dependent instructions, a symbolic expression becomes very large and costly in terms of computational time to analyze practically. In this case, we replace this large symbolic expression with a symbol that represents an expression whose contents are unknown. Our second heuristic relates to the maximum cardinality of the set of facts in our program states. Whenever this set becomes too large, we flush its contents, which is equivalent to symbolically executing a statement whose side effects cannot be known at compile time (e.g., executing a function call whose source code is not available). By giving up some accuracy (i.e., reducing the strength of our static analysis automatically when needed), we improve the runtime of symbolic execution significantly; our analysis takes in the range of milliseconds to a few seconds in all the tests that we will show later in the experimental evaluation section.

3.3 Profile-Guided Memory Partitioning

Profiling has been used in program analysis for various purposes, examples of which include identifying hot regions for mapping to accelerators [41], disambiguating dependences for speculative optimizations [12], and identifying parallel loops [29,32]. In this work, we use profiling for a completely different purpose. Specifically, we use our profiler-based dynamic dependence analysis to perform memory partitioning.

The profiling and trace analysis methods for collecting statistics about a program's runtime behavior are related. Mahapatra et al. [33] analyze the execution traces of programs to assess the impact of compressing the data stored in the memory system and exchanged between the processor and the memory system on the performance, power consumption, and cost of the memory system. However, unlike our work, they do not employ memory partitioning and do not use address compression as a memory design technique.

4 Memory Partitioning

The ultimate goal of our research is to design scalable, parallel memory systems. To achieve this goal, we propose using memory partitioning to decompose the single, unified memory view of a software program into many small memories. This partitioned memory system has two important properties that bring parallelism and

scalability. First, the generated memories operate in parallel, providing higher throughput. Second, they accommodate independent parts of the program address space which eliminates the need for keeping these memories coherent, thereby enabling the use of many memories without excessive overheads for maintaining coherence.

4.1 Accelerator Memory Model

As explained in Sect. 2, most application-specific accelerators use independent address spaces on the host and accelerator, and simplify the host-accelerator coherence problem by prohibiting the use of data structures that are not trivially serialized (i.e., translated into a format for communication and then reconstructed at the receiver side).

In this work, we use a distributed shared memory programming model across the host and the accelerator. The accelerator directly operates using the virtual addresses of the host memory, which is achieved by implementing the accelerator main memory as a cache for the accelerator virtual address space. A memory access on the accelerator can be a miss on the accelerator main memory (i.e., last level cache—LLC), in which case the page or line that encloses the target address is copied from the host main memory to the accelerator LLC and the accelerator proceeds with its operation.

As our host and accelerator share the same address space and there is no address translation mechanism needed, the lower level memories (i.e., caches) generated by memory partitioning must also be accessed using the same addresses. For instance, a typical method that simply promotes an array A to a distinct memory MEM_A and only uses array offsets as the address to this memory may not be applicable, as it is possible for some pointer in the application to also point to the same array (e.g., $p = \&A[0]$) and be used to indirectly access it.

In our case, when we generate a memory MEM_A, it is typically implemented as a cache[5] whose contents are populated from the accelerator LLC, and in turn, from the host memory. The exact same host addresses of A are used when accessing MEM_A. Then, a memory access that uses pointer p containing $\&A[0]$ will also be connected to MEM_A and will directly use the address stored in it to access the desired element in array A.

4.2 Partitioning the Memory of a Procedure

In this work, we apply memory partitioning only to the hottest procedure in the target application, determined *a priori* by our profiler. Since any arbitrary connected code fragment spanning multiple procedures can be converted into the form of a single

[5] In the cache implementation, it is possible to save storage space by identifying duplicate, dead, and constant bits. When a bit within a cache data line (or address) can always be computed as a function of other fixed bits in the same data line (or, respectively, in the same address), no cache hardware resources are needed for that bit. The missing bit can be recreated by recomputing it when needed. Similarly, a bit proved by the compiler to be dead need not be stored as it will never be used, and a constant bit need not be stored as its value can always be predicted with perfect accuracy.

procedure,[6] the algorithms described herein can be applied without loss of generality to arbitrary connected code fragments as well.[7] Let M denote the number of instructions in this procedure that access the memory (i.e., *load* and *store* instructions). By applying memory partitioning, we partition these M memory instructions into N groups such that:

1. If two instructions are dependent, they must be in the same group. For instance, a store to and a load from the same memory location are dependent and must be placed to the same group.
2. The smallest groups of memory instructions, where the groups are not dependent on each other, must be identified. For instance, a store and a load that always reference different memory locations are not directly dependent on each other, and should be placed into different groups, as long as there is no other instruction or set of dependent instructions that is also dependent on both this load and this store.

When these N groups of memory instructions are identified, the address space spanned by these M memory instructions is naturally decomposed into N independent subspaces. Since these subspaces are independent, we can use a separate memory for each group of memory instructions and let the memories operate without any coherence hardware across them.

The most important step in memory partitioning is analyzing the memory references in the input program and determining whether they should be placed into the same group or different groups. To determine this, we pose a data dependence question[8] to find out if any two given memory instructions are dependent.

The pseudocode for our memory partitioning algorithm is given in Fig. 3. In this algorithm, we first construct an undirected dependence graph (V, E), where the vertices $v \in V$ represent the memory instructions in the program and the undirected edges $e \in E$ represent the data dependences between memory instructions. To build this graph, we check the dependences across all pairs of memory instructions. This step has a computational complexity of $O(|V|^2 \times D)$ where $O(D)$ represents the complexity of the dependence tests employed for this purpose. While using complicated tests can have high computational complexity (e.g., the Omega test [38] has exponential worst case complexity), using a profiler-based dependence analysis can reduce the dependence analysis complexity to a hash table look-up operation of $O(1)$ complexity. Once we have the undirected data dependence graph, we use Tarjan's union-find algorithm [46] to find the connected components of this graph, which has a complexity of $O(|V| + |E| \times \alpha(|V|))$, where α is the inverse Ackermann function.

[6] An arbitrary multiple-entry, multiple-exit connected code fragment possibly spanning multiple procedures and files (e.g., the hot basic blocks of an executable) can be converted to a single-entry, single-exit code fragment resembling a procedure, by initially executing a multi-way branch based on the entry point address (program counter), and having all the original exits of the code fragment jump to a single return instruction. One can also use the profiler for discovering the most probable targets of indirect branches, when the target(s) cannot be found statically.

[7] In this work, we assume that the target procedure is either a leaf in the call graph, or it makes calls only to functions with an equivalent hardware version in our component library (e.g., $max(a, b)$).

[8] We provide a definition for *dependence* in Sect. 5.

```
    input  : List of memory instructions in target program fragment,
             Dependence analysis results
    output: Memory Partition Table, Set of generated memories
    // Start with an empty dependence graph
 1  DependenceGraph = empty undirected graph;
    // Vertices of the graph are instructions in the program
 2  for i ∈ Memory Instructions do
 3  |   DependenceGraph.addVertex(i);
 4  end
    // Edges of the graph are dependences across instructions
 5  for i ∈ Memory Instructions do
 6  |   for j ∈ Memory Instructions do
 7  |   |   if instrID(j) > instrID(i) and depends(i, j) ≠ NO then
 8  |   |   |   DependenceGraph.addEdge({i, j});
 9  |   |   end
10  |   end
11  end
    // Find connected components of DependenceGraph
12  components = DependenceGraph.CC();
    // The set of memories generated as a result of partitioning
13  memories = empty set of memories;
14  for comp ∈ components do
    |   // Each component is a new memory
15  |   mpart = empty memory partition;
16  |   mpart.setID(getUniqueMemoryID());
    |   // Instructions in the component are connected to the new memory
17  |   for i ∈ Memory Instructions of comp do
18  |   |   mpart.add(i);
19  |   |   PartitionTable[i] = mpart;
20  |   end
21  |   memories.add(mpart);
22  end
```

Fig. 3 The pseudocode for our memory partitioning algorithm

This algorithm satisfies the goals of memory partitioning: it separates instructions into the maximal number of groups and ensures that instructions in different groups do not have dependences between them. At this point, each connected component represents a subspace of the memory address space that can be implemented as an independent physical memory. We assign a unique memory identifier to each connected component and populate a table that maps memory instruction identifiers to their corresponding memory identifiers. This table is called the memory partition table and will be used by the compiler back end while generating custom hardware for each memory instruction. In the partitioned memory system, instead of accessing one large, common memory, each instruction accesses its corresponding new memory.

Note that our algorithm does not partition program data directly, but instead it partitions the memory instructions in the program which inherently identifies independent subspaces. One can think that memory partitioning using program data can also achieve the same result. For instance, an array A and another array B can simply be allocated to two different memories that are accessed independently. Yet, such a method will not be successful when pointers, dynamic memory allocation, aliasing, and other features that complicate direct partitioning of program data are employed. Unfortunately, these features are commonly observed in most popular general purpose languages such as C/C++. When the same name is used to access distinct variables, or two different names are used to refer to the same variable, or memory allocation

sizes unknown at compile time are used, our approach can still perform dependence analysis and partition memory.

4.3 Hierarchical Memory Partitioning

The performance of the memory system created using memory partitioning depends on the granularity of the memories generated as a result of applying our algorithm. However, it is important to note that in order to create a new memory, the compiler must prove that this new memory and the rest of the memories can operate without the need of coherence hardware across them throughout the *entire* execution of the target software program fragment. Because of this constraint, the maximum memory level parallelism that can be extracted at such a coarse granularity can be limited.

Notice that when analyzing whether two memory instructions are dependent at a given program scope to perform memory partitioning, the compiler must consider all instances of the two memory instructions, which includes all iterations of all the loops enclosing the two instructions in the target program scope. As a result, if memory partitioning is applied at an inner program scope such as an inner loop in a loop hierarchy, then the number of instances of the memory instructions the compiler must examine to identify a possible dependence is reduced. This also reduces the size of the address subspaces spanned by memory instructions and improves the chances that the set of addresses accessed by these two instructions do not overlap, leading to new opportunities for partitioning. In turn, a memory generated as a result of applying our memory partitioning at one program scope can be further partitioned into smaller independent memories at an inner program scope, as shown in Fig. 4. Even if a parent memory belonging to an outer scope does not give rise to more than one child memory in an inner scope, there can be an opportunity to save energy by specializing the child memory (for example, by using a smaller number of address bits in the child memory). By exploiting this property, we can generate a multi-level memory hierarchy where the memories become smaller, faster, and more energy efficient as we move towards the lower levels in the hierarchy (i.e., closer to the leaf nodes).

To apply hierarchical memory partitioning, we refine the memory partitioning algorithm presented in the previous section by making two modifications. First, we modify it to operate at a given particular program scope. A data dependence question defined over a given loop scope assumes the index variables of all enclosing outer loops to be constant (i.e., it checks dependences across a single particular loop iteration, instead of all iterations). Second, instructions connected to different memories (i.e., belonging to different connected components of the dependence graph) at an outer program scope cannot have any dependence between them at the inner scopes, and as a result, we do not re-examine dependences between them at inner loops. We apply this modified algorithm to all scopes of the target program to generate a multi-level memory hierarchy.

Our iterative memory partitioning algorithm, the pseudocode of which is shown in Fig. 5, starts partitioning at the outermost program scope and moves towards inner program scopes. The outermost program scope corresponds to the target procedure to be accelerated and the inner program scopes correspond to the loops in the procedure.

```
int a[N,M], b[N,M]; //N,M powers of two
link MEM1 = MEM0; //MEM1=A[*,*]
link MEM2 = MEM0; //MEM2=B[*,*]
for(int i0=0; i0<N*M; ++i0)
  MEM1[ A[i0/M,i0%M] ] = g(MEM2[ B[i0/M,i0%M] ]);
for(int i1=0; i1<N; ++i1) {
  link MEM3 = MEM1; //initialize child MEM3=A[i1-40,*] from parent MEM1=A[*,*]
  link MEM4 = MEM1; //initialize child MEM4=A[i1,*] from parent MEM1=A[*,*]
  if(test(i1)) {
    for(int j=init(i1); !done(j); j=next(j)) {
      int t = (i1>=40 ? MEM3[ a[i1-40,j] ] : h(j)) & 0x7;
      MEM4[ a[i1,j] ] ^= f(t);
    }
  }
  //flush of read-only MEM3=A[i1-40,*] not needed
  unlink MEM1 = MEM4; //flush changes in MEM4=A[i1,*] into MEM1
}
//flush of read-only MEM2=B[*,*] not needed
unlink MEM0 = MEM1; //flush changes in MEM1 to MEM0
```

Fig. 4 Hierarchical memory partitioning example. The parallel hardware for this code was depicted in Fig. 1. Within any invocation of the `j` inner loop, outer loop index `i1` is a constant, therefore the load from `a[i1-40,j]` and the store/update into `a[i1,j]` are guaranteed not to overlap, and can be assigned to different memories `MEM3` and `MEM4` within this invocation of the `j` loop, respectively. Furthermore, as an energy-saving optimization example, since only 3 bits of the data from `MEM3` is live (because of the `&0x7` operation), and since (assuming N,M are powers of 2) the upper $log_2(N)$ bits of the `MEM3` addresses are constant; a specialized $log2(M)$ by 3 bit wide memory (or a cache for the same) can be constructed to serve as `MEM3`. Iterations of the outer `i1` loop can be executed in parallel and out of order, using a set of replicated `j` loop hardware units each with its own `MEM3` and `MEM4` memories. Through specialized synchronization hardware, iteration `i1` of the outer loop can start as soon as iterations `i1-40,i1-41,`... of the `i1` loop are done, thus ensuring sequential code dependences are correctly respected. Note that without hierarchical memory partitioning, the said load and store/update operations are dependent, and cannot be placed in different memories. Link and unlink instruction semantics are explained in Sect. 4.4

We obtain this program scope ordering by sorting the loops in the target program based on the reverse post order numbers of their loop header basic blocks. We then apply our modified algorithm to each program scope in this order. When a memory is partitioned into child memories, for each child memory, we insert a new child memory node and a new parent-child edge to our memory hierarchy tree, resulting in an N-ary tree. In a manner similar to the single-scope partitioning algorithm, we again construct a memory partition table that marks the instruction to memory mappings as well as a set of memories generated by the whole memory partitioning process.

4.4 Implementation Details and Hardware Implications

Initialization and Flushing of Child Memories. When our compiler partitions memory at a procedure scope, the data contained in the generated memories will be valid during the execution of each invocation the hardware accelerator. These memories need to be initialized from the host memory at the entry to the accelerator and the outputs of the accelerator must be sent back to the host (i.e., flushed to the host memory) upon exiting the accelerator. However, when the compiler generates memories at an inner scope,

```
    input  : List of memory instructions in target program fragment, program CFG
    output: Hierarchical Memory Partition Table,
            Memory Hierarchy Tree, Generated Memories Table
 1  ScopeList = find all program scopes of CFG for memory partitioning;
 2  SortedScopeList = sort ScopeList using entry BB reverse post order numbers;
    // Initialize all scopes to have empty set of generated memories
 3  foreach  Scope S ∈ SortedScopeList do
 4  |   GeneratedMemories[S] = empty set of memories;
 5  end
 6  MEM_0 = getUniqueMemoryID();
    // Memory hierarchy tree is initialized to have a single root node
 7  MemoryHierarchyTree = MEM_0;
    // All memory instructions are initially connected to MEM_0
 8  foreach  Instruction i ∈ CFG.getInstructions() do
 9  |   HierarchicalPartitionTable[i] = MEM_0;
10  end
    // Partition memories at each program scope sequentially
11  foreach Scope S ∈ SortedScopeList do
        // Get instructions in scope S
12      (ChildPartitionTable, ChildMemories) = partitionMemoryAtScope(S);
13      foreach  instruction-memory pair (i, childMEM) ∈ ChildPartitionTable do
            // Get the old memory for i
14          parentMEM = HierarchicalPartitionTable[i];
            // Update with child memory in the hierarchical partition table
15          HierarchicalPartitionTable[i] = childMEM;
16          if childMEM ∉ MemoryHierarchyTree then
                // Save the (parentMEM, childMEM) pair to the memory hierarchy
17              MemoryHierarchyTree.addNode(childMEM);
18              MemoryHierarchyTree.addEdge((parentMEM, childMEM));
19          end
20      end
        // Record the memories generated at this scope
21      foreach  memory childMEM  in ChildMemories do
22      |   GeneratedMemories[scope].add(childMEM);
23      end
24  end
```

Fig. 5 The pseudocode for our hierarchical memory partitioning algorithm

such as over one iteration of an outer loop, these memories must be initialized and flushed at a finer granularity, namely, at each entry to and exit from the target program region (i.e., at each outer loop iteration). To ensure correct execution, our compiler inserts memory initialization and flush operations at the entry and exit points of the body of each loop in the program, as shown in Fig. 6 (and also in Fig. 4). Note that when a child memory is implemented as a cache, the initialization costs are incurred not upfront, but incrementally, as cold misses occur.

Initialization and flushing of child memories introduce additional copying overheads the severity of which must be analyzed by the compiler. If the compiler identifies the copying overheads to be too high to amortize, a child memory should not be created and all corresponding memory instructions should be connected to the next level parent memory.

In some cases, the overheads associated with memory initialization and flushing can be reduced or eliminated. As an example, initialization of a write-only memory, or flushing of a read-only memory can be completely eliminated. Similarly, a memory containing a data structure with local scope is dead upon exit from that scope, and does not need to be flushed and typically does not need to be initialized. On the other

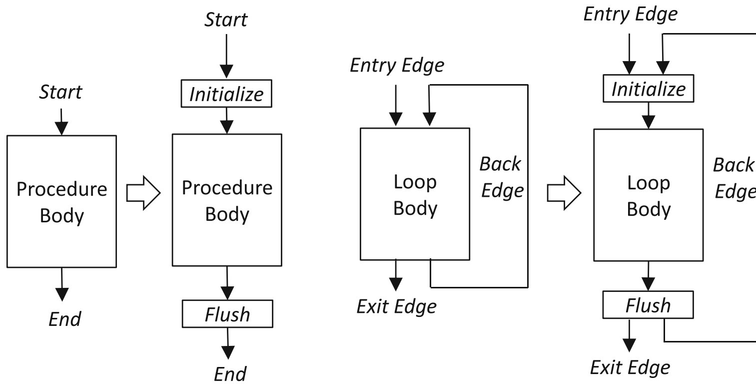

Fig. 6 Inserting initialize and flush instructions to the CFG at procedure entry/exit edges (*left*) and loop body entry/exit edges (*right*)

hand, in some cases, these overheads must be incurred regardless of whether a child memory is generated. For example, an important optimization opportunity for making the initialization and flushing of a child memory free, exists where a subset of the address bits used within a child memory are constant, whereas the same subset of the address bits are variable in the parent memory.

- For instance, to access a data in off-chip DRAM memory, we must first activate the target row and bring it to a row buffer which is similar to a row-size child memory residing inside the DRAM chips. All subsequent accesses to the same row can be served from the row buffer which is faster and consumes less energy than the DRAM array. In this case, if we extract a new intermediate level memory and associate the DRAM row buffer with it, then the initialization/flushing overheads of this memory will not be considered as *extra* since we would have to incur these overheads even without extracting the memory. The DRAM row address bits will remain constant during the region where the child memory (row buffer) is active.
- Similarly, for the case of a one-data-item child memory consisting of a register (a child memory accessed with 0 address bits), loading data from the parent memory into the register at the beginning of a region, operating on the register within the program region, and writing the register back to the memory (if dirty) at the end of the region is how hardware normally works, therefore the initialization and flushing overheads must be incurred anyway. Accessing the register child is the same as accessing a constant address within the parent.
- Similarly, consider a set-associative cache maintaining a set prediction field for each load or store instruction: in this case the predicted cache set of an instruction is the child memory used by this instruction, while the entire cache (comprising all sets) is the parent memory. The set selection bits of the address are constant within the region where the child is active, yielding faster access time and less energy expenditure for the child memory. Overhead related to activating a new child memory must be incurred anyway, as the predicted set of a memory instruction changes.

Eliminating False Sharing Across Memories. It is important to note that although memory partitioning identifies disjoint address subspaces of the application, mapping these address spaces to physical caches with large cache lines (e.g., 32–64 bytes) to exploit spatial locality can lead to false sharing. For instance, two distinct variables `a` and `b` can reside within the same block in the original memory space of the application, which would mean that the same cache line will be duplicated in the memories generated for these variables, namely, `MEMa` and `MEMb`. This would violate our assumption where `MEMa` and `MEMb` can operate independently without coherence, as the order in which these cache lines are flushed to their common parent memory can change the semantics of the program.

To overcome this problem, our caches employ fine granularity dirty bits: we keep one dirty bit for each byte in the cache. When flushing a cache line to its parent cache, only the dirty bytes are written back and the rest of the bytes are discarded. As a result of this optimization, the order in which the two cache lines in sibling memories are flushed to their parent memory is immaterial, as the intersection of their dirty bit sets must be empty per memory partitioning rules. Optionally, for reducing cache traffic, we also add a valid bit per byte, which is initially zero when a child cache line is first referenced, until the byte is written. When an invalid byte is read before it is written within a child memory cache, the line is brought into the child memory cache from the parent memory; otherwise the line is not brought in. The valid bits and the dirty bits features together avoid the unnecessary fetching of a line destined to be overwritten, as well as false sharing. While in a general purpose memory hierarchy, the costs of such dirty and valid bits could be prohibitive (i.e., 2 bits per byte), in a customized application-specific environment, this overhead can be reduced significantly by the compiler. As the compiler extracts memories, it can also analyze the bit-width of the operations performed on each memory and implement dirty and valid bits at that granularity. For instance, when a memory that contains an array of 64-bit integers is generated, this overhead can be reduced to 2 bits per 8-bytes.

Parallel Accesses to a Single Memory. Despite memory partitioning techniques, it is possible for many memory instructions to be connected to the same memory, thus requiring that memory to allow parallel accesses. We apply the following techniques to solve this problem. First, two requests can use the same port to a memory unless they are executed simultaneously in the same cycle. Statically assigning each load/store to one of the interleaved banks of a memory [15] is not applicable within our method, which already generates minimal, indivisible memories: if the loads/stores accessing a memory could be partitioned into N groups, where each group is guaranteed at compile time to be accessing its own disjoint bank of this memory, these loads/stores would not constitute a minimal group of memory operations, contradicting the definition of a memory (see Sect. 4.2). However, simultaneous accesses within the same memory can be implemented through a combination of, e.g., runtime, dynamic bank interleaving, or one of the full-blown coherent cache designs, or a small number of true multiple ports. Memory partitioning ensures that the coherence hardware is confined to one (small) memory. The presence of compiler-generated synchronization between loads/stores which may access the same location, as well as the absence of memory primitives unique to multiprocessors simplify the coherence hardware design.

5 Program Analysis for Memory Partitioning

As described in Sect. 4, in order to partition memory, we must perform data dependence analysis over all memory instruction pairs. Our compiler performs symbolic program analysis to disambiguate memory references and answer dependence questions. When the compile-time symbolic analysis is unable to prove the existence or non-existence a dependence, using a profiling-based dependence analysis can enable better, yet speculative results. In this section, we first give the details of the static program analysis technique we use in our compiler and then describe how our profiler performs dynamic dependence analysis.

5.1 Static Analysis

The dependence question we pose in order to partition memory is defined as follows:

Definition 1 For a given pair of instructions (I_1, I_2) that access memory enclosed by a given program region R, if there exists any execution of R starting with an entry to R and ending with an exit from R, wherein there is at least one memory location accessed by at least one I_1 instance and at least one I_2 instance, then I_1 and I_2 are *dependent*.[9]

To perform this dependence analysis, compilers typically analyze the target program region (or a program region that encloses this region) to obtain symbolic address expressions for the memory instructions and identify whether these symbolic addresses intersect [22]. The accuracy at which these symbolic address expressions can be disambiguated determines the final quality of memory partitioning. Therefore, we use a powerful, path-sensitive program analysis method called symbolic execution to analyze the target program. Since symbolic execution operates on program states, before explaining symbolic execution, we define a program state.

Definition 2 A *program state*, corresponding to a particular point in a target program, represents the information about that program point obtained by analyzing the program. It contains two entities: a path predicate and a set of program facts. The path predicate represents the condition that must be satisfied for an actual program execution to reach that point in the program. The set of program facts represents all address-value relationships[10] known to hold when an actual program execution reaches this particular program point.

[9] This definition includes input dependences. Excluding input dependences (requiring at least one of I_1 or I_2 to be a store operation) can result in a partitioned memory where more than one copy of a data can exist in the system. However, this case occurs only for data that is exclusively accessed using load instructions (i.e., read-only data), in which case there is still no need for maintaining coherence across memories.

[10] As already summarized in Sect. 3.2, existing symbolic execution methods [21] define a program fact as a mapping from variable names to their values, such as (var, val) mapping variable var to its value val. We modified this definition to record program facts involving variables without names and pointers, such as (ptr, val) which maps the location pointed by ptr to val. We can still capture variable to value mapping information, albeit at a slightly modified form, such as $(\&var, val)$, where $\&var$ is the address expression for variable var.

```
    input  : Program Control Flow Graph
    output: Symbolic Execution Data Flow Results
    // Initial symbolic execution pass over the program fragment
 1  for b ∈ BasicBlocks do
        // Join symbolic program states on incoming edges of b
 2      join(b);
        // Symbolically execute instructions in basic block b
 3      execute(b);
 4  end
 5  repeat
 6      fixpoint = true;
 7      for b ∈ BasicBlocks do
            // Join symbolic program states on incoming edges of b
 8          join(b);
 9          if b is a loop header then
                // Find induction variables of loop starting at b
10              newInductionVars = identify new induction variables of b;
11              if |newInductionVars| > 0 then
                    // Fixpoint reached if no loops with new induction vars
12                  fixpoint = false;
13              end
14          end
            // Symbolically execute instructions in basic block b
            // only if the input program states could have changed
15          if !fixpoint then
16              execute(b);
17          end
18      end
19  until fixpoint = true;
```

Fig. 7 The pseudocode for our symbolic execution algorithm. Please see the "Appendix" for more details of the subroutines `execute` and `join`

For instance, a program state can have $x > 0$ as its path predicate and $(\&y, 0)$, $(\&z, 1)$ as its set of facts. This means that for an actual execution of this program to reach the corresponding program point, the input x must have a value greater than zero. Further, if any execution reaches this program point, the values of variables y and z (i.e., the values at memory locations $*(\&y)$ and $*(\&z)$) must be 0 and 1, respectively.

Iterative Symbolic Execution Symbolic execution of a procedure (or any part of a procedure), given in Fig. 7, works as follows. We execute a pass over the entire input procedure by symbolically executing the instructions in all basic blocks in the CFG[11]. This operation starts with an empty program state (i.e., a program state with no facts and *true* path predicate) corresponding to the start point of the entry basic block of the procedure. As we symbolically execute the instructions in the first basic block, the program state is modified to record the effect of executing those instructions. When we finish executing the last instruction in this basic block, we obtain the program states that correspond to the exit edges of the basic block, and continue execution with the next basic block. Before starting symbolic execution of each basic block, the algorithm joins the program states on all incoming CFG edges (i.e., on all exit edges

[11] The basic blocks are traversed based on their reverse post ordering numbers.

of the predecessor basic blocks) to identify the program state at the entry of the basic block.

Note that a single symbolic execution pass may not be sufficient to identify all induction variables (variables which are set in one iteration of the loop and then used in a future iteration of the same loop) in the procedure. Without identifying all induction variables, an address expression we find for a memory instruction may be incomplete and mislead the dependence analysis to think that the corresponding memory instruction operates on a smaller range of addresses. This can result in incorrect dependence analysis.

To address this induction variable detection issue, we use a fixpoint algorithm that re-executes the target procedure symbolically until all induction variables are identified.[12] Since the symbolic value of an induction variable can change at every loop iteration, the symbolic value expression of induction variables will be different at loop entry edge and loop back edge program states. Therefore, in order to detect loop induction variables, we compare the program states at the loop entry and back edges. When we identify an induction variable, we insert a new mapping of the form $(addr, iv)$ to the entry edge program state of the corresponding loop, where $addr$ represents the address expression of the induction variable and iv is a unique symbolic value expression for this induction variable which represents all possible values the variable can take during all iterations of the enclosing loop. Then, in the rest of the symbolic execution iteration, all subsequent instructions in the program that use this induction variable will refer to this unique symbolic value expression. For instance, when we identify a variable x to be an induction variable and assign it the symbolic value iv_x, we will later discover that the array access $A[x]$ actually refers to a range of values $A[iv_x]$, not a single element of the A array.

Each symbolic execution iteration tries to find more induction variables. In each iteration, a basic block needs to be symbolically re-executed only if the program state at its entry edge changed since the previous iteration. Such a change can only occur if some new induction variables were identified.

Induction Variable Substitution The goal of symbolic execution is to accurately identify the address expressions of memory instructions in the input program. When a fixpoint is reached, we use program states to get a mapping from each memory instruction to the corresponding symbolic address expression. However, these address expressions cannot be used directly for dependence analysis as they do not include the closed-form solutions of induction variables. For instance, assume that we identified store instruction to access an address with symbolic expression $A[iv]$, where iv is a symbolic value expression of an induction variable that is used as an index when accessing array A. Note that we do not have the actual value expression for iv yet, and must conservatively assume that $A[iv]$ can refer to any location in the array A. To find the closed-form solutions for these induction variable symbolic value expressions (i.e., ivs) [25], we use pattern matching. If an induction variable matches a known

[12] Termination of this algorithm can simply be proven using the fact that a program can have a finite number of induction variables. The maximum number of iterations done by our algorithm depends on the length of the longest induction variable dependence chain in the target procedure.

pattern, we replace it with the corresponding solution in all address expressions. For instance, one pattern is defined as follows:

1. If iv is an induction variable of a loop with loop index I,
2. If the symbolic value expression of iv is $init$ at loop entry edge program state,
3. If the symbolic value expression of iv is $iv + step$ at the loop back edge program state,
4. Then, the closed-form solution expression for iv is $init + I \times step$.

Using simple induction variable patterns, we can solve almost all induction variables in typical programs.[13] Once we have these solutions, we can use them to rewrite the address expressions for memory instructions. For instance, an address expression $A[iv]$ can now become $A[2 \times I + 1]$, which is ready for applying dependence analysis[14].

Dependence Tests Once the symbolic address expressions are ready, we can perform dependence analysis. Given a pair of memory instructions (I_1, I_2), with address expressions (A_1, A_2), we perform a set of simple, fundamental tests at this stage to prove/disprove dependences.

1. At least one of I_1 or I_2 must be a store operation to have a dependence (only if we are ignoring input dependences in memory partitioning).
2. If the address expressions A_1 and A_2 are exactly identical, then these expressions refer to the same memory location.
3. If it is possible to prove that the two instructions access variables with distinct names by inspecting the two address expressions (e.g., $A + \text{offset}_1$ and $B + \text{offset}_2$), then there is no dependence. Note that, as a result of path-sensitive symbolic execution, an address expression can refer to multiple locations. For instance, an address expression can be of the form $cond\ ?\ (A + \text{offset})\ :\ (B + \text{offset})$, which means that it refers to array A or B based on the value of variable $cond$ at runtime. In this case, we find the sets of variables accessed by each memory instruction and perform a set intersection. If the intersection set is empty, then there is no dependence.
4. If the user has indicated that the program follows ANSI aliasing rules, we analyze the data types of the address expressions. We use the type information existing in the symbol table[15] to check whether the two instructions access variables of different types. If so, then per ANSI aliasing rules, we assume that there is no dependence. For instance, two stores $p[I_1]$ and $q[I_2]$ are independent if p is a pointer to `int` and q is a pointer to `float`.
5. We also apply other standard dependence tests such as the range test [9] and the GCD test [5]. These tests further improve the accuracy of dependence analysis.

The user can also improve the results by providing the compiler with options that give information about argument aliasing. By indicating that the pointer arguments to

[13] Solving induction variables of inner loops before outer loops enables us to solve higher order induction variables.

[14] We also solve loop exit conditions to find the symbolic loop trip counts, which correspond to the maximum values for loop index variables (i.e., I_{max} values where $0 \leq I < I_{max}$).

[15] In our implementation, we use the symbol information embedded into the assembly files by *gcc* when the `gstabs+` option is used.

procedures cannot be aliased, the user can help the compiler perform better symbolic program analysis and better memory disambiguation.

5.2 Dynamic Dependence Analysis

In some cases, static dependence analysis may not be able to prove whether two memory instructions in the target program are dependent or not. This can occur due to two main reasons. First, the dependence tests used in the compiler may either not be strong enough to prove independence, or may be skipped due to taking too much time, considering compilation time restrictions. Second, the existence of a dependence may depend on a runtime condition that may or may not occur at runtime. In both cases, the compiler must be conservative and assume there is a dependence. A profiler that performs dynamic dependence analysis would not suffer from either problem.

To perform dynamic dependence analysis, we need our profiler to record which set of instructions access the same location, and hence, by Definition 1, are dependent. For this purpose, our profiler keeps a table that maps the memory addresses accessed by the application to the set of instructions that made the accesses. The target procedure in the application is instrumented such that after each memory instruction, we also update the corresponding entry in this table using the unique identifier of the memory instruction. When the application terminates, we analyze the entries in this table, checking the set of instructions that access each memory address. Using this information, we build a dependence graph such that any two instructions that reside in the same set for at least one memory address have a dependence edge between them.

There are certain time and space overheads in dynamically maintaining this table that maps memory addresses to set of instruction identifiers. From the time perspective, for each memory instruction in the original program, we now perform a number of additional memory accesses to update the table. While using a hash table we can make these look-up and insert operations in $O(1)$ time, we still need a set insert operation, a typical implementation of which requires $O(log(|S|))$ operations where $|S|$ is the cardinality of the set (i.e., the number of instructions that accessed the same memory address). In practice, only a very small subset of instructions in the procedure reside in the same set, giving an average complexity of $O(1)$. However, the profiler still needs to make a number of memory accesses at least 3–4 times the original application, which can result in a significant increase in execution time.

From the space overhead point of view, for each address accessed by the original application, we now keep an extra set of instruction identifiers. This space overhead is $O(|S|)$, which can significantly increase the memory space used by the profiler. For instance, if all addresses are accessed by 4 instructions, then at least 4X extra memory space will be used by the profiler.

To reduce these time and space overheads associated with dynamic dependence analysis, we use sampling in our profiler, as already explained in Sect. 2.2. We generate two versions of the target procedure: an instrumented version and an original non-instrumented version. We execute the profiled version of for a time period T_1 and then jump to original version to execute it for a time period T_2, giving a duty cycle of $T_1/(T_1+T_2)$. By applying sampling, both the time and the space overheads of keeping

dynamic dependence information in our profiler is reduced in proportion to the duty cycle.

One can ask the question whether the benefits of maximal memory partitioning based on the dynamic dependence analysis described here may be achievable in practice. Indeed this limit can be approached, provided that errors due to risky speculative memory separation decisions (i.e., there exist memory operations I_1 and I_2 which (1) have never been observed to depend on each other at runtime, (2) are not proved to be independent by static analysis, (3) are speculatively linked to different memories, and (4) do refer to the same location on a new rogue program input) are rare, and are detected through extra overlap detection hardware not on the critical path. When such an error is detected, one simple approach is to abort the current hardware acceleration and revert to the original un-accelerated software version of the hardware, before the changes to the software application memory are flushed back by the hardware accelerator. The next FPGA compilation, or the next release of the ASIC chip, can respect the offending dependence, and correct the error.

6 Experimental Evaluation

We perform our experimental evaluation in two steps. First, we analyze the accuracy of our compiler-based dependence analysis by comparing it with the dependence analysis results obtained from profiling. We then use these dependence results in memory partitioning and show the quality of the resulting partitioned memory system. Before presenting our results, we provide some definitions and elaborate on our target benchmarks.

6.1 Metrics used in Evaluation

We evaluate the quality of our compiler-based dependence analysis using two metrics: static accuracy and dynamic accuracy. Let $d_c(I_1, I_2, R)$ and $d_p(I_1, I_2, R)$ denote the dependence analysis responses given by the compiler and the profiler over a program region R for the instruction pair (I_1, I_2), respectively. Then, we define the function $u(I_1, I_2, R)$ as follows:

$$u(I_1, I_2, R) = \begin{cases} 1, & \text{if } d_c(I_1, I_2, R) = \text{may-alias and } d_p(I_1, I_2, R) = \text{no-alias} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

This function has the value 1 when the compiler is unable to prove that a dependence between I_1 and I_2 does not exist, although it is not actually observed at runtime. Then, we define the static accuracy and the dynamic accuracy of a dependence analysis as follows.

Definition 3 Static accuracy of a compiler-based dependence analysis on a given program measures the fraction of dependence questions that are (1) answered by the compiler-based dependence analysis as *may-alias* and (2) answered by the profiler-based dependence analysis as *no-alias*. The mathematical formula for static accuracy

is as follows.

$$StaticAccuracy(R) = 1 - \frac{\sum_{I_1} \sum_{I_2 > I_1} u(I_1, I_2, R)}{\sum_{I_1} \sum_{I_2 > I_1} 1} \tag{2}$$

$$= 1 - \frac{\sum_{I_1} \sum_{I_2 > I_1} u(I_1, I_2, R)}{(N-1) \cdot N/2}. \tag{3}$$

where N denotes the number of memory instructions in program region R.

Definition 4 Dynamic accuracy of compiler-based dependence analysis on a given program is a modified form of static dependence accuracy that also takes into account the number of times each instruction is executed as a weight function.

$$DynamicAccuracy(R) = 1 - \frac{\sum_{I_1} \sum_{I_2 > I_1} u(I_1, I_2, R) \cdot Weight(I_1, I_2, R)}{\sum_{I_1} \sum_{I_2 > I_1} Weight(I_1, I_2, R)}, \tag{4}$$

where the weight function is calculated using the execution counts of the two instructions over the region R in each dependence question obtained from the profiler:

$$Weight(I_1, I_2, R) = max(count(I_1, R), count(I_2, R)). \tag{5}$$

While we use these static and dynamic accuracy values to determine the quality of our compiler-based dependence analysis, these need not directly translate into the quality of memory partitioning. In a well-partitioned memory system, the individual memories will comprise a smaller number of distinct addresses and therefore a smaller number of data values located at these addresses. Furthermore, if some addresses within a memory are referenced more often than others, there exists an opportunity to encode the addresses of this memory with even less bits. To quantify these reductions, we use two metrics: the number of address bits needed to access the entire target memory and the Shannon entropy [42], defined as follows.

Definition 5 The number of address bits to cover the entire set X of unique locations covered by an access sequence:

$$Address\ Bits(X) = log_2(|X|) \tag{6}$$

Definition 6 Shannon entropy provides a limit on the best possible lossless compression of any communication. For a given distribution of messages $X = \{x_1, \ldots, x_n\}$ to be communicated between a sender and a receiver, and a probability mass function of outcome x_i given as $p(x_i)$, entropy is calculated using the formula:

$$Entropy(X) = -\sum_{i=1}^{n} p(x_i) \cdot log_2(p(x_i)) \tag{7}$$

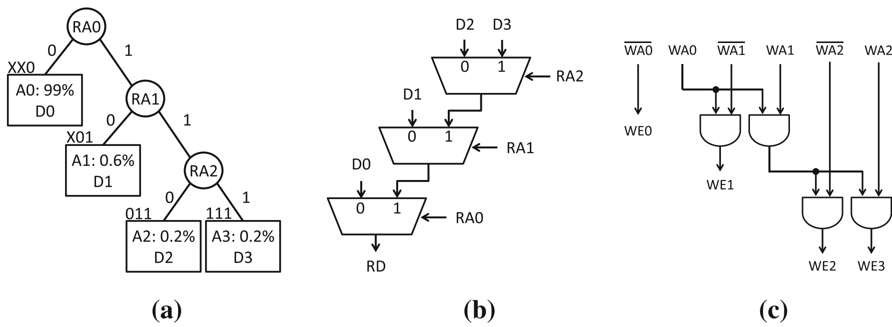

Fig. 8 An example used as a proof of concept for entropy encoding of memory accesses. **a** An entropy-encoded (Huffman) representation of original addresses A0,A1,A2,A3, containing data items D0,D1,D2,D3 respectively, **b** data read multiplexer of memory with entropy-encoded addresses (RA=read address, RD=read data), and **c** write decoder of memory with entropy-encoded addresses (WA=write address, WEi=write-enable for Di)

In our case, we consider the distribution of the addresses in the accesses to each memory to calculate its *address entropy*. This gives the limit on the minimum average number of address bits needed to implement this memory, assuming a perfect encoding is adopted in the compiler and the memory hardware. Entropy is intuitively also a theoretical limit on the average number of logic levels in accessing (reading or writing) a memory, assuming the probability distribution of addresses is known a priori, and can be achieved by accessing more frequent addresses with fewer logic levels, and infrequent addresses with more logic levels, using encoding of addresses within the program itself and within the memory, coupled with speculation and recovery techniques.

Figure 8 gives a proof of concept for entropy encoding of memory addresses over an example. Huffman encoding [28] attempts to approximate entropy, e.g., given four data items D3,D2,D1,D0, located respectively at addresses A3 (occurring 0.2 % in the software execution trace), A2 (occurring 0.2 %), A1 (occurring 0.6 %), and A0 (occurring 99 %) in the original memory, we can Huffman-encode the original addresses as A0=XX0, A1=X01, A2=011, A3=111 as in Fig. 8(a), following a low-endian format for addresses (with address bit 0 being in the least significant position). Here, X means a “don’t care” value. We then can create a memory read multiplexer that reads data item D0 with a delay of only one 2 to 1 multiplexer level, faster than D1, D2, or D3, as shown in Fig. 8b. Thus, the read address (RA) to read data (RD) propagation delays will be faster for data items that are closer to the root of the Huffman-encoded tree, and slower for data items that are more distant from the root of the Huffman-encoded tree. We can also create a Huffman-encoded write decoder to generate the write enable signals WE0,WE1,WE2,WE3, for data items D0, D1, D2, D3, respectively during a write operation, as shown in Fig. 8c. This write decoder also generates the write enable signal for D0 faster than for the other data items D1, D2, and D3. To avoid any glitches in the WE1-WE3 signals at the instant WE0 becomes high (active) at an early time, the decoder circuit could use, e.g., CMOS domino logic [31]. All write enable signals will normally be low (inactive) when the decoder is precharged. Then,

when the decoder is evaluated, one and only one write-enable signal will become high (active) in a glitch-free fashion. As in the case of the read circuitry, the write address to write-enable propagation delay will be faster for data items that are closer to the root of the Huffman-encoded tree, and slower for data items that are more distant from the root of the Huffman-encoded tree.

However, Shannon entropy can do better than Huffman encoding: for example the number of logic levels for the given read multiplexer is at least one 2 to 1 multiplexer logic level, whereas the Shannon entropy of this address execution trace (highly biased to be `A0`) is only about 0.09. In this case, for example, when a particular test in the Huffman tree is highly biased toward 0, an optimized version of the hardware Huffman tree can be built by speculatively assuming the said bit is always 0, while checking if it really is zero and falling back to the original Huffman tree after a few extra clocks, if the speculation was incorrect. In this case, a register normally containing `D0` can be tied directly to the output for reading and will be available as the read data immediately, with zero logic levels, for reading the memory most of the time. Although the practical implementation of such an encoding is the subject of future research in compiler techniques for encoding values of variables, and future application-specific cache hierarchy design and layout, measuring entropy gives us an idea of how close we are to an ideal memory design. Note that a compressed encoding based on the probability distribution of the set of data items read or written in a memory can also give rise to beneficial reductions in the memory data width, although data bit width optimization is beyond the scope of this paper.

When the accesses are evenly distributed across the target memory addresses, the *address bits* value and the entropy values become close. However, if some addresses are accessed much more frequently than other addresses, then encoding can improve the average number of bits needed to perform the communication of addresses between the processing unit/finite state machine/program and memory, and therefore entropy becomes smaller. We compare the two metrics to reveal more information about the underlying memory access patterns.

When we extract a number of memories for an application, using the number of memories extracted to determine the quality of memory partitioning may be misleading, because some memories may simply be accessed only a few times, whereas most of the accesses may be concentrated on only a few memories. To identify the effective number of memory partitions, we use the dynamic access counts of individual memories obtained from profiling and reuse the entropy concept.

Definition 7 For a given set of memories $Y = \{y_1, \ldots, y_n\}$ where $p_m(y_i)$ denotes the empirical probability of accessing memory y_i:

$$EffectiveNumberOfMemories(Y) = 2^{-\sum_{i=1}^{n_{memories}} p_m(y_i) \cdot log_2(p_m(y_i))} \quad (8)$$

gives the effective number of memories extracted from the application. If the memory accesses are distributed across all memories evenly at runtime, then the effective number of memories gets close to the actual number of memories extracted. However, if only one memory is accessed at runtime and all other memories are idle, then this metric approaches one, indicating that one memory dominates the system.

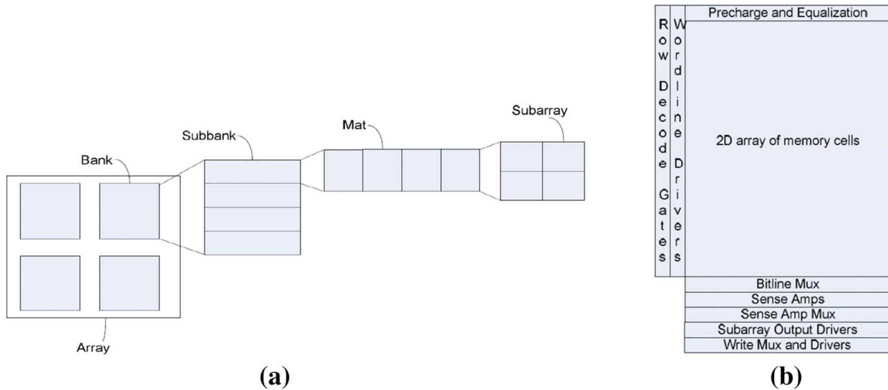

Fig. 9 **a** High level organization of a cache and **b** subarray internals; taken from CACTI technical report [35]

6.2 Modeling Cache Latency and Energy

As a result of profiling the application after memory partitioning, we obtain the number of address bits and address entropy values for all the memories we generated, which identify the optimum size needed for this memory (without encoding and with encoding). However, it is important to identify the exact relationship between the size of a memory and its access latency and energy. To achieve this, we use CACTI [35] which is a tool that models cache area, latency, and timing by considering various internal structures. We now briefly describe this model and how we build a relationship between cache capacity and cache latency/energy.

Caches are internally organized as hierarchical storage units, as shown in Fig. 9a [35]. A cache contains set of banks that can be accessed independently. Each bank has a number of subbanks in it, where only one subbank is accessed at a time. A subbank access is distributed to multiple mats inside the subbank and each mat access is further redirected to one of many subarrays inside it. Fig. 9b shows the organization of a single subarray.

There are many components in a cache, but only a few of them are responsible for most of the access latency and energy. One such component is the combination of row decoder and wordline drivers. The row decoder takes a row address and generates an enable signal for only the corresponding row. This enable signal is propagated as the wordline enable signal to all cells in the target row, which requires driving many access transistors. Therefore, dedicated wordline drivers are used to increase the strength of this signal. Second, the SRAM cells are connected to a number of bitlines that connect them to the sense amplifiers. Since there is a connection from each row to each bitline, the length of each bitline is essentially proportional to the number of rows in the cache. As the cache gets larger, these bitlines also get longer and make a larger contribution to latency and energy. Once the data is sensed by the sense amplifiers in a subarray of the cache, the subarray output drivers are used to send the data read from the array to the external interface of the entire cache. It is important to note that this communication between the subarrays and external cache interface must be very wide to transfer an

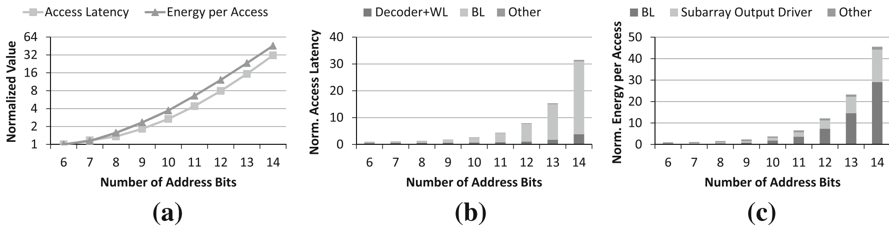

Fig. 10 **a** Scaling of cache latency and energy, **b** scaling of individual cache access latency components, and **c** scaling of individual cache energy components. The x-axis denotes the number of bits ($addr_bits$) in the address that are used as index bits. Cache capacity is proportional to 2^{addr_bits}

entire cache line quickly. While other components in a cache also add to the latency and energy of the cache, their contribution is relatively smaller.

Figure 10 (left) shows the relationship between the number of index bits in a cache (where cache capacity is proportional to 2^{addr_bits}) and two factors: access latency and dynamic energy per access. For very small caches, latency and energy increase slowly with capacity, but as the cache becomes larger, doubling the capacity also doubles latency and energy. As can also be seen in Fig. 10 (center and right), the decoders, wordline drivers, and bitlines dominate the access latency in large caches. Especially long bitlines become a bottleneck in the performance of the cache. Similarly, the dynamic energy of the cache is also dominated by two factors: bitlines and subarray output drivers. In this case, in addition to the bitlines, the dynamic energy cost of driving the data from the subarray to the output edges of the cache becomes very costly as the cache gets larger. We use these CACTI-based latency and energy models in our overall average latency and average dynamic energy per access computations. Using these models, we translate our results expressed in terms of number of address bits used to access caches into physical metrics (i.e., latency and energy) related to the memory hierarchy.

6.3 Benchmarks

In our evaluation, we use benchmarks from the SPEC CPU 2000 and CPU 2006 benchmark suites [27,45]. For each benchmark, we identified the hottest procedure shown in Table 1 and inlined all calls to user functions within that procedure. We are using a set of eight benchmarks from these suites which have a hot function that takes a large fraction of the execution time, that does not have recursive function calls, and does not suffer from code explosion by inlining. In this table, we also report the number of instructions in our RISC-like intermediate representation and the number of loops in these procedures after inlining.

6.4 Compiler-Based Dependence Analysis Results

The first set of results we obtain are related to the accuracy of the static program analysis in our compiler. Based on definitions 3 and 4 given in Sect. 6.1, we evaluated

Table 1 Benchmarks used in evaluation and their hottest functions targeted by our compiler for memory partitioning

Bench. no	Bench. name	Hottest procedure	Instr.	Loops	Static acc. (%)	Dynamic acc. (%)
183	equake	smvp	413	2	94	95
401	bzip2	mainGtU	625	1	94	95
403	gcc	approx_reg_cost	268	4	89	88
429	mcf	primal_bea_mpp	339	4	69	59
433	milc	mult_su3_na	134	2	97	96
458	sjeng	setup_attackers	410	5	94	91
470	lbm	LBM_performStreamCollide	864	1	95	97
473	astar	wayobj::makebound2	426	2	97	91
Average					92	91

The last two columns give the static and dynamic accuracy of compiler-based dependence analysis

Table 2 *Compiler-based* (conservative) memory partitioning results: average address entropy and average number of address bits without partitioning and with partitioning

Bench.	w/o Partitioning		w/ Partitioning		Memories	Effective Num.
	Entropy	Bits	Entropy	Bits	(Scalar/Array)	of Memories
equake	10.8	15.9	7.5	7.8	30 (26/4)	9.66
bzip2	9.3	16.6	5.3	5.3	30 (28/2)	15.20
gcc	5.0	5.5	0.7	0.7	26 (25/1)	9.16
mcf	11.9	12.3	9.2	9.4	20 (13/7)	6.00
milc	6.2	6.9	2.2	2.2	24 (10/14)	15.65
sjeng	6.2	6.5	3.2	3.2	15 (14/1)	8.25
lbm	9.3	15.2	3.0	3.0	103 (64/39)	75.34
astar	6.4	10.9	3.0	4.9	38 (33/5)	20.58

The next column gives the number of scalar and array memories generated by applying partitioning, and the last column presents the effective number of memories (see Definition 7)

the static and dynamic accuracy of the dependence analysis methods we use. In both metrics, the dependence analysis results of the compiler analysis is compared against the dependence analysis results reported by the profiler. For static accuracy, we record the dependences that cannot be resolved by our compiler, but are never encountered during the profiling run. We then calculate the ratio of these cases to the total number of all possible dependence questions. In case of dynamic accuracy of the same set of compiler questions and compiler answers, we also take into account how many times each instruction is executed and use these values as the weights to dependences. Our results in the last two columns of Table 1 indicate that our compiler is able to successfully answer more than 90 % of the dependence questions asked, in both static and dynamic accuracy metrics for almost all benchmarks.

6.5 Memory Partitioning Results and Discussion

We now present our memory partitioning results. These results are obtained by performing dependence analysis using the compiler-based and profiler-based methods given in Sect. 5. We then compare these two sets of results and discuss the underlying reasons for any similarities and differences we observe.

Compiler-based (conservative) Memory Partitioning Results. Table 2 shows two pairs of results for each benchmark. Each pair reports the average address entropy and the average number of address bits used by the application when accessing the memories in the memory system. The first pair of values correspond to the case when no memory partitioning is used, i.e., all memory instructions on the accelerator access a single, unified memory. The second pair of values correspond to the case when we partition the memory of the program when mapping it into custom hardware. The results under the columns (address) “bits” and “entropy” show the average number of logic levels in reading and writing the memories. The results in column “bits” is the average number

of logic levels with a flat address encoding that assigns a unique binary number to each unique original address. If there are n items accessed in the original memory, the number of logic levels under the "bits" column will be $log_2(n)$. The column named "entropy" shows the results with a memory whose read multiplexer and write decoder have been entropy-encoded according to perfect a priori knowledge of the probability distribution of the original addresses. With entropy encoded addresses, the average number of logic levels in accessing a memory with n elements will generally be in the range 0 to $log_2(n)$, inclusive. For both the baseline case and the partitioned cases, and in either the "bits" or the "entropy" column, the cost of initialization from or flushing to the next memory is not included. The number of data items to be initialized and/or flushed in both cases is equal to 2^{addr_bits}, where *addr_bits* is the number given under the "bits" column in the baseline case.

By examining the differences in the number of address bits needed with and without memory partitioning (i.e., pairs 1 and 2 in Table 2), we observe that there is a significant reduction in the results for all benchmarks. This indicates that the average size of the ideal memories accessed with memory partitioning is significantly lower. An interesting observation follows from the comparison of the entropy and the number of address bits results with and without memory partitioning. When there is no memory partitioning, the entropy values are typically much lower than the address bits, indicating that the accesses made to the single, unified memory are quite non-uniformly distributed. On the other hand, when we partition memory, the difference between entropy and the number of address bits becomes much smaller. This indicates that by partitioning memory, independent access patterns were routed to distinct memories, leading to more uniform access patterns to be received by the generated child memories.

Figure 11 shows the reduction in the average number of logic levels during reading or writing, the average memory access latency and the average energy per memory access for all the benchmarks. The number of logic levels during reading or writing are taken from Table 2 "bits" (flat encoding) and "entropy" (entropy-based encoding) columns. The other two sets of results (b and c) are obtained by accumulating the latency and energy of each memory access served by the new memories created by memory partitioning, respectively. The number of accesses to each memory is obtained by using the memory access statistics from the second profiling run (see Fig. 2). The results are then normalized against the average latency and energy values obtained without memory partitioning, i.e., our baseline. Analyzing the results without entropy encoding, we observe that by applying compiler-based memory partitioning, we achieve a 20–85 % reduction in the average number of logic levels and 70–99 % reduction in average access latency and dynamic energy. This means that, for these benchmarks, we can eliminate more than 70 % of the dynamic energy consumed by the memory system, and serve the memory accesses 3.3X–100X faster. Making the assumption that memories with entropy encoded addresses can be designed as efficiently as normal memories, our results with entropy encoded memories are also seen to be mostly in the same range; they also show very significant improvements. Note that the VLSI layout of memories with entropy encoded addresses is a future research topic, and making a precise estimate is difficult at this time.

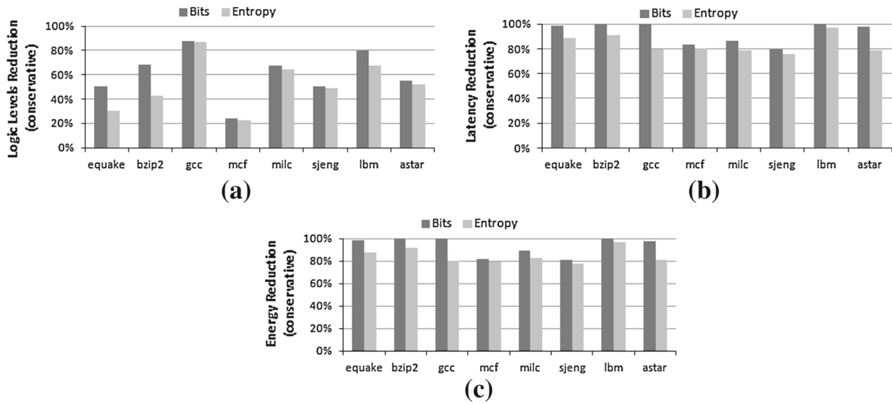

Fig. 11 Reduction results (computed as $(value_{old} - value_{new})/value_{old}$, where "old" means "without memory partitioning" and "new" means "with memory partitioning") obtained using *compiler-based* memory partitioning: **a** average logic levels during reads or writes, **b** est. average access time, and **c** est. average dynamic energy, with flat encoding ($log_2(n)$ bits) and entropy encoding of addresses

Table 3 *Profiler-based* (optimistic) memory partitioning results: average address entropy and average number of address bits without partitioning and with partitioning

Bench.	w/o Partitioning		w/ Partitioning		Memories	Effective Num.
	Entropy	Bits	Entropy	Bits	(Scalar/Array)	of Memories
equake	10.8	15.9	5.9	6.1	59 (25/34)	33.01
bzip2	9.3	16.6	4.1	4.1	84 (28/56)	39.28
gcc	5.0	5.4	0.1	0.1	39 (39/0)	12.15
mcf	11.9	12.3	8.8	8.9	32 (22/10)	8.51
milc	6.2	7.0	2.2	2.2	24 (10/14)	15.44
sjeng	6.1	6.4	2.2	2.2	35 (32/3)	21.61
lbm	9.6	15.3	3.3	3.3	105 (66/39)	76.89
astar	6.4	10.7	1.8	1.8	61 (33/28)	25.80

The next column gives the number of scalar and array memories generated by applying memory partitioning, and the last column presents the effective number of memories (see Definition 7)

Profiler-based (optimistic) Memory Partitioning Results. Our second set of memory partitioning results are obtained by using profiler-based dependence analysis. Since the profiler results are valid for only one particular input and the profiler might miss some dependences due to sampling, the partitioning results from this analysis are optimistic. This means that they represent the limit that no static analysis-based partitioning can ever outperform. The results shown in Table 3, when compared to the equivalent results with compiler-based partitioning given in Table 2, verify this expectation. First of all, the number of memories generated by applying profiler-based memory separation is always higher than that obtained using compiler-based analysis. This means that the profiler can rule out some dependences that did not occur at runtime, which results in the generation of more connected components in the dependence graph. Further,

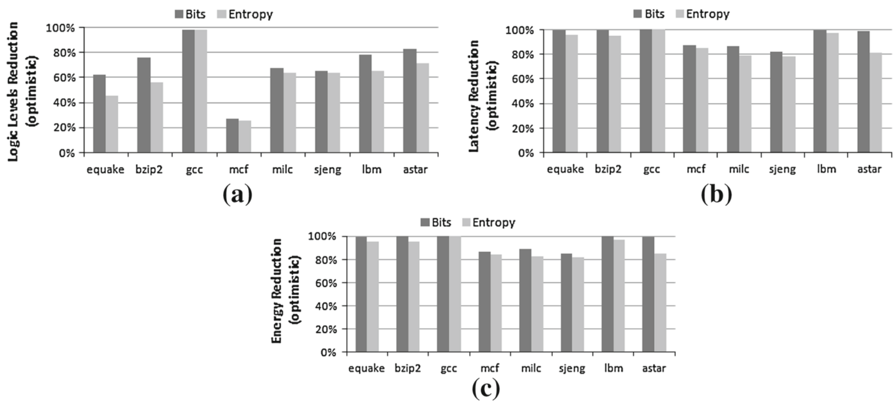

Fig. 12 Reduction results (computed as $(value_{old} - value_{new})/value_{old}$) obtained using *profiler-based* memory partitioning: **a** average logic levels during reads or writes, **b** est. average access time, and **c** est. average dynamic energy, with flat encoding ($log_2(n)$ bits) and entropy encoding of addresses

all average entropy and number of address bits results obtained with profiler-based memory partitioning are either equal to or better than compiler-based partitioning, which indicates that smaller memories were generated as a result of using profiler-based dependence analysis.

Figure 12 shows the reduction in average memory access latency and average energy per memory access for all benchmarks, using profiler-based memory partitioning. By applying profiler-based memory partitioning, we again achieved up to 99 % reduction in access latency and dynamic energy. We observe that our final results with profiler-based partitioning are typically very close to those obtained with compiler-based partitioning. This indicates that the energy and latency benefits we obtained with our compiler-based approach are very close to the ideal results, realized by our profiler-based approach.

Multi-Level Hierarchical Memory Partitioning Results. Finally, we present our multi-level hierarchical memory partitioning results. In *bzip2* benchmark, there is only one level of scoping hierarchy inside the hottest procedure where we focus, and, therefore, we excluded it from the tables and figures that we present in this subsection. Table 4 shows the details of the all average entropy and number of address bits results obtained with multi-level memory partitioning. Here, we give results for both with and without flushing overhead. Note that the results with no flushing overhead are in agreement with our previous findings. However, when we consider that the flushing operation of child memory contents to their parent memories is something that we have to pay anyway, the results without this flushing overhead looks much more promising.

Figure 13 shows the reduction in average memory access latency and average energy per memory access using multi-level memory partitioning. Although, we expect and also observe much more accesses to multi-level memories due to excessive flush overheads in some benchmarks, the energy and latency figures are still better compared to the single memory model. This anomaly can be easily explained by the help of Fig. 13a. The amount of logic level reduction is quite high on the average across

Table 4 *Compiler-based* (conservative) *Multi-level* memory partitioning results: average address entropy and average number of address bits without partitioning and with partitioning (* denotes results with flushing overhead (X) and without flushing overhead (Y) given in (X/Y) format)

Bench.	w/o Partitioning		w/ Partitioning		Memories	Effective Num.
	Entropy	Bits	Entropy*	Bits*	(Scalar/Array)	of Memories*
equake	8.7	12.0	(5.7/1.0)	(5.9/1.0)	63 (40/23)	(25.5/18.6)
gcc	6.5	7.8	(1.1/1.1)	(1.1/1.1)	28 (25/3)	(13.5/13.3)
mcf	11.8	12.2	(0.4/0.3)	(0.4/0.3)	28 (19/9)	(12.2/7.1)
milc	11.0	13.6	(6.1/0.6)	(6.3/0.7)	40 (26/14)	(30.1/16.0)
sjeng	7.7	9.3	(2.2/1.0)	(2.2/1.0)	19 (10/9)	(11.1/11.1)
lbm	11.4	12.8	(3.0/0.0)	(3.0/0.0)	142 (103/39)	(8.9/2.2)
astar	7.0	13.9	(0.7/0.7)	(1.3/0.7)	42 (37/5)	(22.7/20.0)

The next column gives the number of memories generated by applying memory partitioning, and the last column presents the effective number of memories

all simulated benchmarks, and lower average entropy and number of address bits indicate that the size of each memory partition can be arranged to accommodate only a few number of addresses. This type of size reduction in memories directly implies considerable amount of energy and latency reduction as shown in these figures.

Discussion of Results. While the results presented in Tables 2, 3, 4 and Figs 11, 12, 13 give an idea of the overall improvement in the memory system, we also zoom inside target benchmarks and identify the reasons for their behavior when memory partitioning is applied.

Our first observation is that for *mcf*, *sjeng*, *astar*, and *bzip2* benchmarks, there is a non-negligible difference in the number of address bits results when applying compiler-based and profiler-based memory partitioning. When we analyzed *mcf*, we observed that due to extensive use of pointers in the application, our present static analysis was unable to track the most recent value of a pointer reused towards the middle of the target procedure, failing to disambiguate it. This is also the reason why our dependence analysis accuracies for *mcf* in Table 1 are lower than the other benchmarks. On the other hand, by using the type of the pointer, our compiler was still able to extract a separate memory for that type of variables, reducing the overall impact of this inaccuracy in the final partitioning results. The profiler was easily able to identify the set of locations accessed by this pointer and eliminate the associated dependence edges. Our analysis with *sjeng* revealed that most memory accesses were successfully disambiguated. However, since our compiler currently does not have a monotonicity analysis, we were unable to prove that the two memory instructions $I1$ and $I2$ in the code fragment shown in Fig. 14(left) can never access the same memory address. Third, in *astar*, the type analysis pass in our compiler was unable to identify that one pointer accesses a subfield of a structure, whereas another pointer accesses another subfield of the same structure, which was the cause for profiler-based partitioning outperforming compiler-based partitioning. Finally, in *bzip2*, we observe that the profiler is in fact extracting information that is impossible for the compiler

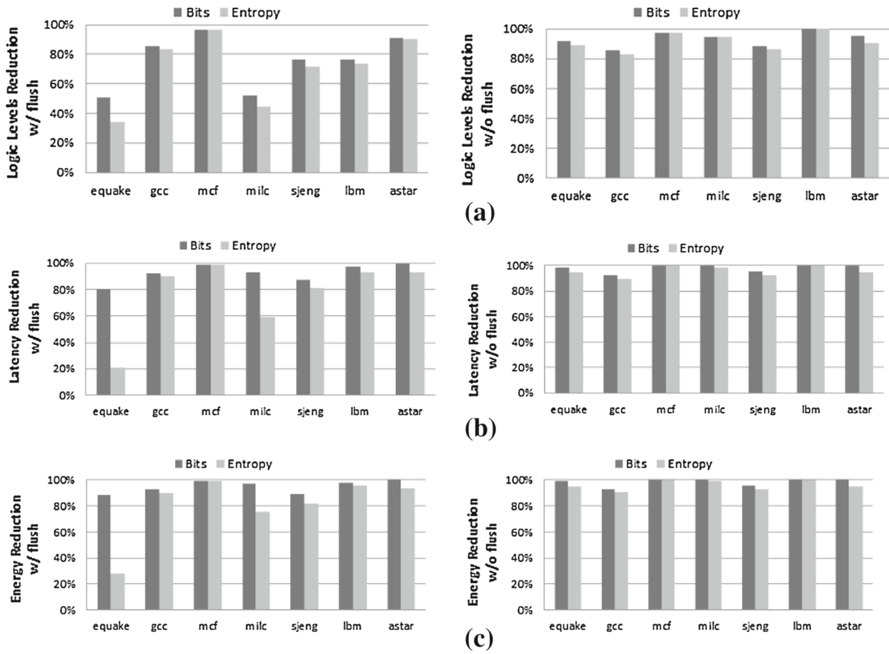

Fig. 13 Reduction results (computed as $(value_{old} - value_{new})/value_{old}$) obtained using *multi-level* memory partitioning: **a** average logic levels during reads or writes with (on the left) and without flush overhead (on the right), **b** est. average access time with and without flush overhead, and, finally, **c** est. average dynamic energy with and without flush overhead, with flat encoding ($log_2(n)$ bits) and entropy encoding of addresses

to obtain at compile time. We repeat a simplified version of this code fragment in Fig. 14(right). Unless the programmer provides a hint to the compiler indicating that the actual values of parameters $i1$ and $i2$ passed to the target hot procedure are separated by a large value, the compiler cannot match the profiler results (note that we consider loads that refer to the same location to be dependent in the context of this paper).

We also analyzed the remaining tests and observed that for *milc* and *lbm*, enabling the GCD test in our compiler significantly improved the memory partitioning results. In *milc*, this test enabled us to disambiguate memory accesses `a->e[i][0]`, `a->e[i][1]`, and `a->e[i][2]` (obtained after loop unrolling), when the index variable `i` increments in steps of 3 (i.e., `i=i+3`). In *lbm*, which is similar to a 19-point stencil computation, the GCD test simply proved all 19 stencil points to be non-overlapping.

7 Conclusion

We proposed a memory partitioning method to automatically generate application-specific memory systems. This method can generate a multi-level memory hierarchy that does not suffer from the scalability issues of existing memory systems by eliminating coherence across memories at the same level of the hierarchy. The only requirement

```
void setup_attackers(...) {
  ...
  for(i) {
     if(cond1) {
       see_attackers[0][numw].piece = ...; //I1
       numw++;
    }
  }
  //no change to numw here
  for(j) {
    if(cond2) {
       see_attackers[0][numw].piece = ...; //I2
       numw++;
    }
  ...
}
```

```
bool mainGtU( UInt32 i1, UInt32 i2,
    UChar* block, UInt32 nblock...) {
  ...
  Int32 k = nblock + 8;
  do {
    c1 = block[i1]; c2 = block[i2];
    if( c1 != c2 ) return (c1 > c2);
    i1++; i2++;
    c1 = block[i1]; c2 = block[i2];
    if( c1 != c2 ) return (c1 > c2);
    i1++; i2++;
    ...
    k -= 8;
  }while(k>=0);
  ...
}
```

Fig. 14 Simplified code fragments from *sjeng* (*left*) and *bzip2* (*right*). In *sjeng*, due to lack of monotonicity analysis in our compiler, we were unable to prove that `I1` and `I2` can never access the same memory location. In *bzip2*, the compiler has no information about the difference of parameters $i1$ and $i2$ passed to this procedure, and therefore cannot prove that two ranges $[i1_{start}, i1_{end}]$ and $[i2_{start}, i2_{end}]$ do not overlap in practice. The profiler, on the other hand, can identify that this dependence never occurs at runtime

for ensuring correct execution of the target application is the initialization and flushing of memories at the entry and exit points of the program regions these memories are extracted from.

We demonstrated a compiler that uses this method to generate application-specific memory and showed a compiler analysis that can be used to obtain answers to the dependence analysis questions posed by this method. The effectiveness of the proposed method is evaluated by using a profiler-based strategy. We demonstrated a profiler-based calculation of the average memory access latency and average dynamic energy per access with and without our compiler-based memory partitioning method. Finally, we compared our compiler-based results with an oracle, profiler-based memory partitioner that represents the limit that can be achieved by memory partitioning.

Our results show that the average latency and energy of accessing memory system can be dramatically reduced by employing our memory partitioning method. Further, in most cases, our compiler-based partitioner can achieve results very close to the profiler-based results, which proves that our approach is very close to achieving optimum memory partitioning.

We believe that automatic memory partitioning can be effective in solving the memory scalability problem, enabling the realization of custom, application-specific memories. In this work, we also established several topics for future research.

Appendix: Details of Symbolic Execution

Basic Block Execute Algorithm

Figure 15 presents the pseudocode for the `execute` function referred to in Fig. 7. Given a basic block b, this function starts with the entry program state of b, executes all instructions in the basic block, and then generates one or more exit program states.

```
    input  : Basic block b entry program state
    output: Basic block b exit program states
 1  State state = b.entryState;
 2  for instruction i ∈ b do
 3      switch i.type do
            // arithmetic operation
 4          case ADD:
 5              Expr e0 = state.lookup(i.src[0]);
 6              Expr e1 = state.lookup(i.src[1]);
 7              Expr result = simplify(ADD, e0, e1);
 8              state.update(i.dst[0], result);
 9              break;
10          end
            // memory load operation
11          case LOAD:
12              Expr e0 = state.lookup(i.src[0]);
13              Expr result = state.lookup(e0);
14              state.update(i.dst[0], result);
15              break;
16          end
            // branch on condition register
17          case BR:
18              Expr cond = state.lookup(i.src[0]);
19              Expr condn = simplify(NOT, cond);
20              b.exitStates[0].facts = state.facts;
21              b.exitStates[1].facts = state.facts;
22              Expr p1 = simplify(AND, state.pred, cond);
23              Expr p2 = simplify(AND, state.pred, condn);
24              b.exitStates[0].pred = p1;
25              b.exitStates[1].pred = p2;
26              break;
27          end
            // Cases for other operations
28          ...
29      endsw
30  end
```

Fig. 15 Symbolic execution `execute` algorithm for executing the instructions in a basic block

Executing each instruction in the basic block has some side effects on the program state, updating the stored symbolic value expressions of some registers or memory locations recorded in the program state. The figure shows three types of instructions. The first one is an arithmetic operation which is handled by reading the value expressions corresponding to the two source registers, creating a simplified result expression, and updating the value of the corresponding destination register in the program state with this simplified result. The second one is a memory load instruction which starts with extracting the value expression of the source register (containing the operand address) from the program state, and then requires a lookup in the program state with this extracted value expression, which is treated as an address expression. Finally, the lookup result is used to update the value of the destination register. The third operation is a two-way branch the direction of which depends on the value a condition register. If the basic block ends with a two-way conditional branch, the symbolic value expression of the condition register obtained from the program state and its logical negation are used to update the path predicate expressions on the two outgoing edges of this basic block. Other types of instructions are variants of these three types of instructions and are omitted for brevity.

```
   input  : Program states on (forward) CFG edges coming into basic
            block b: inState1, inState2
   output: Entry program state of basic block b: outState
   // Combine input predicates
1  outState.pred = simplify(OR, inState1.pred, inState2.pred);
   // Add facts from first edge
2  for fact f1 ∈ inState1.facts do
3  |   Expr val1 = f1.value;
4  |   Expr val2 = inState2.lookup(f1.address);
5  |   Expr val = simplify(f1.pred ? val1 : val2);
6  |   outState.addFact(f1.address, val);
7  end
   // Add facts from second edge
8  for fact f2 ∈ inState2.facts do
   |   // Handled in first loop?
9  |   if f2.address exists in inState1.facts then
10 |   |   continue;
11 |   end
12 |   Expr val1 = inState1.lookup(f2.address);
13 |   Expr val2 = f2.value;
14 |   Expr val = simplify(f2.pred ? val2 : val1);
15 |   outState.addFact(f2.address, val);
16 end
```

Fig. 16 Symbolic execution `join` algorithm for joining the program states corresponding to two incoming forward CFG edges to a basic block

Program State Join Algorithm

Figure 16 gives the pseudocode for the `join` function referred in Fig. 7 Given a basic block *b*, this function takes two program states on two (forward) CFG edges coming into *b*, and computes the program state that is valid at the entry point of *b*. When there are more than two incoming edges to a basic block, this function is called more than once (e.g., `join(edge1, join(edge2,edge3))` for three incoming edges). The path predicate at the entry of *b* is obtained by performing a logical *or* operation on the incoming edge path predicates. For each address-to-value expression mappings on the first program state, the corresponding value expression on the second program state is obtained and a conditional expression is created.[16] The second loop handles the address-to-value expression mappings on the second program state that do not exist on the first program state (i.e., the address expressions not handled by the first loop). As a result of applying this join operation, the program state at the entry of basic block *b* is obtained, which is the input to the `execute` function given in Fig. 15. It is important to note that this algorithm is used only for joining forward edges. Backward edges define loops, which are handled by the algorithm in Fig. 7.

[16] If the values on both states are the same, the expression simplification algorithm eliminates the conditional expression, i.e., `simplify(cond ? x : x) = x`.

References

1. Anderson, Jennifer M., Amarasinghe, Saman P., Lam, Monica S.: Data and computation transformations for multiprocessors. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95, pp. 166–178, New York, NY, USA. ACM (1995)
2. Anderson, Jennifer M., Lam, Monica S.: Global optimizations for parallelism and locality on scalable parallel machines. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93, pp. 112–125 (1993)
3. Avissar, Oren, Barua, Rajeev, Stewart, Dave: An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Trans. Embed. Comput. Syst. **1**(1), 6–26 (2002)
4. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: Proceedings of the Tenth International Symposium on Hardware/software codesign, CODES '02, (2002)
5. Banerjee, U.K.: Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell (1988)
6. Baradaran, Nastaran, Diniz, Pedro C.: A compiler approach to managing storage and memory bandwidth in configurable architectures. ACM Trans. Des. Autom. Electron. Syst. **13**(4), 61:1–61:26 (2008)
7. Benini, L., Macchiarulo, L., Macii, A., Poncino, M.: From architecture to layout: partitioned memory synthesis for embedded systems-on-chip. In: Proceedings of Design Automation Conference, 2001, pp. 784–789 (2001)
8. Benini, L., Macii, A., Poncino, M.: A recursive algorithm for low-power memory partitioning. In: Proceedings of the 2000 International Symposium on Low Power Electronics and Design, ISLPED '00, pp. 78–83, ACM (2000)
9. Blume, W., Eigenmann, R.: The range test: a dependence test for symbolic, non-linear expressions. In: Supercomputing '94, (1994)
10. Bobda, Christophe: Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications, 1st edn. Springer, New York (2007)
11. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05, pp. 519–538 (2005)
12. Chen, T., Lin, J., Dai, X., Hsu, W.-C., Yew, P.-C.: Data dependence profiling for speculative optimizations. In: Evelyn Duesterwald (ed.) Compiler Construction, vol. 2985 of Lecture Notes in Computer Science, pp. 57–72 (2004)
13. Cimitile, A., De Lucia, A., Munro, M.: Qualifying reusable functions using symbolic execution. In: Proceedings of the Second Working Conference on Reverse Engineering (1995)
14. Coen-Porisini, A., De Paoli, F., Ghezzi, C., Mandrioli, D.: Software specialization via symbolic execution. IEEE Trans. Softw. Eng. **17**(9), 884–889 (1991)
15. Cong, J., Jiang, W., Liu, B., Zou, Y.: Automatic memory partitioning and scheduling for throughput and power optimization. In: Computer-Aided Design—Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on, pp. 697–704 (2009)
16. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering (2008)
17. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '03, pp. 15–24, Washington, DC, USA. IEEE Computer Society (2003)
18. Ebcioğlu, K., Altman, E.R.: Daisy: dynamic compilation for 100% architectural compatibility. In: Proceedings of the 24th Annual International Symposium on Computer Architecture, ISCA '97, pp. 26–37, New York, NY, USA. ACM (1997)
19. El-Ghazawi, T., Cantonnet, F.: Upc performance and potential: a NPB experimental study. In: Proceedings of the 2002 ACM/IEEE Conference On Supercomputing, Supercomputing '02, pp. 1–26 (2002)
20. Elkarablieh, B., Godefroid, P., Levin, M.Y.: Precise pointer reasoning for dynamic test generation. In: ISSTA '09: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (2009)

21. Fahringer, T., Scholz, B.: A unified symbolic evaluation framework for parallelizing compilers. IEEE Trans. Parallel Distrib. Syst. **11**(11), 1110–1125 (2000)
22. Feautrier, Paul: Dataflow analysis of array and scalar references. Int. J. Parallel Program. **20**(1), 23–53 (1991)
23. Gokhale, Maya B., Graham, Paul S.: Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays, 1st edn. Springer, New York (2010)
24. Gokhale, M.B., Stone, J.M.: Automatic allocation of arrays to memories in fpga processors with multiple memory banks. In: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '99 (1999)
25. Haghighat, M., Polychronopoulos, C.: Symbolic program analysis and optimization for parallelizing compilers. In: Banerjee, Utpal, Gelernter, David, Nicolau, Alex, Padua, David (eds.) Languages and Compilers for Parallel Computing, Volume 757 of Lecture Notes in Computer Science, pp. 538–562. Springer, Berlin (1993)
26. Heinrich, J.: Origin and onyx2 theory of operations manual, silicon graphics corporation. Document number 007-3439-002, (1997). http://techpubs.sgi.com/library/manuals/3000/007-3439-002/pdf/007-3439-002.pdf
27. Henning, J.L.: Spec cpu2006 benchmark descriptions. SIGARCH Comput. Arch. News **34**(4), 1–17 (2006)
28. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)
29. Ketterlin, A., Clauss, P.: Profiling data-dependence to assist parallelization: framework, scope, and optimization. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '12 (2012)
30. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976)
31. Krambeck, R.H., Lee, C.M., Law, H.-F.S.: High-speed compact circuits with cmos. IEEE J. Solid-State Circuits **17**(3), 614–619 (1982)
32. Larus, J.R.: Loop-level parallelism in numeric and symbolic programs. IEEE Trans. Parallel Distrib. Syst. **4**(7), 812–826 (1993)
33. Mahapatra, Nihar R., Liu, Jiangjiang, Sundaresan, Krishnan, Dangeti, Srinivas, Venkatrao, Balakrishna V.: A limit study on the potential of compression for improving memory system performance, power consumption, and cost. J. Instr. Level Parallelism **7**, 1–37 (2005)
34. Moon, Soo-Mook, Ebcioğlu, Kemal: Parallelizing nonnumerical code with selective scheduling and software pipelining. ACM Trans. Program. Lang. Syst. **19**(6), 853–898 (1997)
35. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: CACTI 6.0. http://www.hpl.hp.com/research/cacti/. (2009)
36. Necula, G.C.: Translation validation for an optimizing compiler. SIGPLAN Not. **35**(5), 83–94 (2000)
37. Numrich, Robert W., Reid, John: Co-array fortran for parallel programming. SIGPLAN Fortran Forum **17**(2), 1–31 (1998)
38. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, pp. 4–13, ACM (1991)
39. Rinard, M.C., Diniz, P.C.: Commutativity analysis: a new analysis framework for parallelizing compilers. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1996)
40. Rogers Jr, Hartley: Theory of Recursive Functions and Effective Computability. MIT Press, Cambridge (1987)
41. Rul, S., Vandierendonck, H., De Bosschere, K.: Towards automatic program partitioning. In: Proceedings of the 6th ACM Conference on Computing frontiers, CF '09 (2009)
42. Shannon, Claude E., Weaver, Warren: A Mathematical Theory of Communication. University of Illinois Press, Champaign (1963)
43. Silberman, Gabriel M., Ebcioglu, Kemal: An architectural framework for supporting heterogeneous instruction-set architectures. Computer **26**(6), 39–56 (1993)
44. So, B., Hall, M.W., Ziegler, H.E.: Custom data layout for memory parallelism. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04 (2004)
45. Standard Performance Evaluation Committee. Spec cpu2000 benchmarks, (2000). http://www.spec.org/cpu2000/

46. Tarjan, Robert Endre: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
47. Weinhardt, M., Luk, W.: Memory access optimization and ram inference for pipeline vectorization. In: FPL (1999)
48. Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S.L., Hargrove, P., Hilfinger, P., Husbands, P., Lancu, C., Kamil, A., Nishtala, R., Su, J., Welcome, M., Wen, T.: Productivity and performance using partitioned global address space languages. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, PASCO '07 (2007)
49. Yelick, K., Hilfinger, P., Graham, S., Bonachea, D., Su, J., Kamil, A., Datta, K., Colella, P., Wen, T.: Parallel languages and compilers: perspective from the titanium experience. Int. J. High Perform. Comput. Appl. **21**(3), 266–290 (2007)