COMPILER-BASED MEMORY OPTIMIZATIONS FOR HIGH PERFORMANCE COMPUTING SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Emre Kültürsay

© 2013 Emre Kültürsay

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2013
The dissertation of Emre Kültürsay was reviewed and approved* by the following:

Mahmut Kandemir  
Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee  

Padma Raghavan  
Professor of Computer Science and Engineering  

Mary Jane Irwin  
Professor of Computer Science and Engineering  

Qian Wang  
Associate Professor of Mechanical and Nuclear Engineering  

Kemal Ebcioğlu  
Global Supercomputing Corporation  
Special Member  

Lee Coraor  
Associate Professor of Computer Science and Engineering  
Graduate Officer of the Department of Computer Science and Engineering  

*Signatures are on file in the Graduate School.
Abstract

Parallelism has always been the primary method to achieve higher performance. To advance the computational capabilities of state-of-the-art high performance computing systems, we continue to rely on increasing parallelism by putting more processors into systems and integrating more cores and more logic into the processors. However, the returns from increasing parallelism are diminishing. Putting more chips, more cores, more logic already started to bring less and less improvements in performance. The primary cause of this behavior is memory. The scalability problem of memory systems translates into a discrepancy between the increase in processor performance and the improvements in memory bandwidth, latency, and energy efficiency. The memory systems in high performance computers are no longer able to provide data to the parallel computational units at a fast enough rate, with a low enough latency and energy. This memory problem plagues the design of high performance systems, and scaling trends show that managing and accessing memory efficiently is one of the most crucial challenges for realizing exascale systems.

This dissertation focuses on compiler-based methods to address these memory issues. Specifically, various compiler-based memory optimizations to improve the memory behavior of application-specific and general purpose high performance computing systems are proposed and evaluated.

The first part of this dissertation identifies the memory scalability problem in the design of application-specific hardware accelerators and proposes a compiler-based automatic memory partitioning method to address this issue. This method generates energy efficient, high bandwidth, low latency memory systems and enables the generation high performance accelerators that can scale up to a huge number of custom-designed chips.

The second part of this dissertation targets general purpose systems and attacks the memory latency and bandwidth problem in many-core processors that are used as building blocks for general purpose high performance computing systems. It presents compiler generation of software prefetching and streaming store instructions and shows their effectiveness in hiding long memory latencies and saving precious memory bandwidth on a cutting edge many-core processor.
# Table of Contents

List of Figures vii

List of Tables xi

Acknowledgments xii

## Chapter 1

**Introduction** 1

1.1 Definition of a Supercomputer ......................... 2
1.2 Supercomputer Architecture .......................... 2
1.3 Types of Supercomputers .............................. 3
1.4 Increasing Parallelism for Higher Performance ........ 6
1.5 When Parallelism is Not Enough: The Memory Problem ... 8
  1.5.1 The Memory Problem in Application Specific Systems . 9
  1.5.2 The Memory Problem in General Purpose Systems .... 10
1.6 Compiler-based Optimizations to Address Memory Problems . 12

## Chapter 2

**Automatic Synthesis of Scalable Memory Systems for Application-Specific Hardware Accelerators** 14

2.1 Introduction ......................................... 15
2.2 Overview of the Proposed System ....................... 18
  2.2.1 Application-Specific Hardware Accelerators and the Role of Automatic Memory Partitioning .......... 18
  2.2.2 Infrastructure .................................... 23
2.3 Related Work ......................................... 26
  2.3.1 Compiler-Based Memory Partitioning ................ 27
2.3.2 Symbolic Execution in Compilers ................................ 30
2.3.3 Profile-Based Memory Partitioning .............................. 32
2.4 Memory Partitioning .................................................. 32
  2.4.1 Accelerator Memory Model ...................................... 33
  2.4.2 Partitioning the Memory of a Procedure ....................... 34
  2.4.3 Hierarchical Memory Partitioning .............................. 38
  2.4.4 Implementation Details and Hardware Implications .......... 41
2.5 Program Analysis for Memory Partitioning ...................... 46
  2.5.1 Static Analysis .................................................. 46
  2.5.2 Dynamic Dependence Analysis ................................ 55
2.6 Experimental Evaluation ............................................ 57
  2.6.1 Metrics used in Evaluation .................................... 57
  2.6.2 Modeling Cache Latency and Energy ......................... 63
  2.6.3 Benchmarks ..................................................... 67
  2.6.4 Dependence Analysis Results ................................. 67
  2.6.5 Memory Partitioning Results and Discussion ............... 68
2.7 Conclusion .......................................................... 76

Chapter 3
Latency Hiding and Bandwidth Saving Memory Optimizations
for General Purpose Many-core Processors .......................... 77
3.1 Introduction ......................................................... 78
3.2 Background ......................................................... 80
  3.2.1 Architectural Overview of the Intel Xeon Phi Coprocessor 80
  3.2.2 Programming and Optimizing for the Intel Xeon Phi Copro-
       cessors .......................................................... 84
  3.2.3 Intel Composer XE 2013 ...................................... 87
3.3 Prefetching on the Intel Xeon Phi Coprocessor ................. 88
  3.3.1 Software Prefetching .......................................... 88
  3.3.2 Hardware Prefetching .......................................... 93
  3.3.3 Combined Software/Hardware Prefetching .................... 94
3.4 Streaming Store Instructions on the Intel Xeon Phi Coprocessor 94
  3.4.1 Types of Streaming Store Instructions ....................... 95
  3.4.2 Cache Line Eviction Instructions ............................. 97
  3.4.3 Controlling the Compiler Streaming Store and Cache Line
       Eviction Behavior ............................................. 97
3.5 Evaluation and Results ........................................... 99
  3.5.1 Target Applications .......................................... 99
  3.5.2 Target System Specifications ............................... 100
  3.5.3 Results ..................................................... 101
3.6 Related Work ........................................ 108
3.7 Conclusion ........................................... 110

Chapter 4
Summary and Future Work ................................ 111
4.1 Summary of Dissertation Contributions .......... 111
4.2 Future Research Directions ........................... 112

Bibliography .............................................. 114
List of Figures

1.1 As of November 2012, the Titan supercomputer is ranked the highest performance supercomputer in the world according to the top500 list. It contains about 200 cabinets, each cabinet housing 24 blades.

1.2 A blade of Titan contains 4 nodes that communicate with other blades using Cray’s proprietary Gemini [1] interconnect (the two large chips on the left). When combined, the four nodes on a blade contain four AMD Opteron 6274 [2] processors with 16 cores each (the four large chips arranged vertically at the center), 32GB DDR3 memory (16 vertical DIMMs on the left), four Nvidia Tesla K20X GPUs [3] (four chips/heat sinks on the right), 24GB GDDR5 memory to be used by the GPUs (two vertical DIMMs at the center), and four chips that convert hyper transport interface of Opteron CPUs to PCI-E interface of Tesla GPUs (four small gray heatsinks on the right-middle).

1.3 The RAMP [4] multi-FPGA reconfigurable system designed for emulation of multiprocessors systems.

1.4 Organization of a BEE3 board [5] that is used in the RAMP system. Each board has four Virtex-5 FPGAs connected using a ring interconnect, 4 GbE and 8 CX4 connections for communicating across boards, and up to 64GB of DDR2 memory on 16 DIMMs.
2.1 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 Figure 2.4.

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

2.3 The pseudocode for our memory partitioning algorithm.

2.4 Hierarchical memory partitioning example. The parallel hardware for this code was depicted in Figure 2.1. 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 \( \log_2(M) \) 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 Section 2.4.4.

2.5 The pseudocode for our hierarchical memory partitioning algorithm.

2.6 Inserting initialize and flush instructions to the CFG at procedure entry/exit edges (left) and loop body entry/exit edges (right).
2.7 The pseudocode for our symbolic execution algorithm. Please see the appendix for more details of the subroutines execute and join. 48
2.8 Symbolic execution **execute** algorithm for executing the instructions in a basic block. 49
2.9 Symbolic execution **join** algorithm for joining the program states corresponding to two incoming forward CFG edges to a basic block. 50
2.10 An example used as a proof of concept for entropy encoding of memory accesses. (a) An entropy-encoded (Huffman) representation of original addresses $A_0, A_1, A_2, A_3$, containing data items $D_0, D_1, D_2, D_3$ 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, WE=write-enable for Di). 61
2.11 (a) High level organization of a cache and (b) subarray internals; taken from CACTI technical report [6]. 64
2.12 (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 ($\text{addr} \cdot \text{bits}$) in the address that are used as index bits. Cache capacity is proportional to $2^{\text{addr} \cdot \text{bits}}$. 66
2.13 Reduction results (computed as $(\text{value}_{\text{old}} - \text{value}_{\text{new}})/\text{value}_{\text{old}}$) 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. 70
2.14 Reduction results (computed as $(\text{value}_{\text{old}} - \text{value}_{\text{new}})/\text{value}_{\text{old}}$) 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. 73
2.15 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 $I_1$ and $I_2$ can never access the same memory location. In **bzip2**, the compiler has no information about the difference of parameters $i_1$ and $i_2$ passed to this procedure, and therefore cannot prove that two ranges $[i_1_{\text{start}}, i_1_{\text{end}}]$ and $[i_2_{\text{start}}, i_2_{\text{end}}]$ do not overlap in practice. The profiler, on the other hand, can identify that this dependence never occurs at runtime. 75
3.1 Organization of the Intel Xeon Phi Coprocessor, taken from [7]. 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Details of a single core in the Xeon Phi Coprocessor, taken from [7].</td>
<td>83</td>
</tr>
<tr>
<td>3.3</td>
<td>Results with benchmarks that benefit from software prefetching.</td>
<td>102</td>
</tr>
<tr>
<td>3.4</td>
<td>Results with benchmarks that are insensitive to software prefetching.</td>
<td>102</td>
</tr>
<tr>
<td>3.5</td>
<td>Results with benchmarks that improve using streaming stores.</td>
<td>107</td>
</tr>
<tr>
<td>3.6</td>
<td>Results with benchmarks that are insensitive to or degrade using streaming stores.</td>
<td>107</td>
</tr>
<tr>
<td>3.7</td>
<td>Comparison with an Intel Xeon processor based system.</td>
<td>108</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Benchmarks used in evaluation and their hottest functions targeted by our compiler for memory partitioning. The last two columns give the static and dynamic accuracy of compiler-based dependence analysis. 67

2.2 Compiler-based (conservative) memory partitioning results: average address entropy and average number of address bits without partitioning and with partitioning. 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). 69

2.3 Profiler-based (optimistic) memory partitioning results: average address entropy and average number of address bits without partitioning and with partitioning. 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). 72

3.1 Target parallel applications. 100

3.2 Target system specifications. 101
Many times I have been asked the question of why I have been pursuing a computer science and engineering doctorate degree, and in the first couple of years of my PhD studies, my answer was “I want to learn more in this topic”. Then, I noticed that the accumulation of technical knowledge was just a side effect. The main skills I was building were to learn on my own, think out of the box, and pose and solve problems nobody ever thought before. I believe that this process transformed me from a student/learner into a researcher/engineer/teacher. Throughout this transformation, I received great support and coaching from my academic advisor Mahmut Kandemir. Starting from the very first day, for all the six years I spent at Penn State, he always treated me with the utmost respect, professionalism, and sincerity; I am extremely grateful for all his guidance and feedback.

When I take a step back and look my PhD life, I clearly see that the most important turning point was to start working with Kemal Ebcioğlu. Dr. Ebcioğlu shared his years of experience with me, helping me grasp various engineering problems and solve them practically. It was a huge privilege for me to work with him and his wisdom guided me through all the years we collaborated; I am forever indebted to him.

An engineering PhD student frequently finds himself wondering about the after life: “Am I going to continue in academia or the industry after graduation?”. On the academic side, while I was actively involved in research, I also wanted to get a taste of teaching. I was very lucky to be have chance to teach a Computer Architecture course under the supervision of Vijaykrishnan Narayanan. It was an unmatched experience for me, and I thank him deeply for sharing all the great teaching methods he has built over years of experience with me, and helping me uncover my passion for teaching. I would also like to express my thanks towards Mary Jane Irwin, Padma Raghavan, Anand Sivasubramaniam, and Chita Das who profoundly influenced me during my PhD life and shaped my academic career.

The other side of the picture was the industry, and my experience with industrial product cycle was also very limited. I was blessed with the opportunity
to become a member of the Intel Compilers team. I had the chance to work on compiler development and testing for a fascinating new product. It was a unique experience, and made me comprehend how excited I get when I work with cutting-edge technology. I am very grateful to my manager Rakesh Krishnaiyer for making me a part of the team, for working with me on our projects, and for enlightening me with his experience.

Throughout my PhD studies, I had the honor of collaborating with many outstanding researchers and engineers. I sincerely thank Gürhan Küçük, Ravi Narayanaswamy, Rajiv Deodhar, Hideki Saito, Serguei Preis, Robert Davies, Onur Mutlu, Yuan Xie, Suman Datta, and Suzanne Shontz for their valuable contributions to and constructive comments on my works.

Special thanks to Karthik Swaminathan for all his hard work in our co-authored papers. I sincerely appreciate the technical discussions and collaborations I had with my labmates Onur Kayıran, Mahmut Aktasoğlu, Umut Orhan, Myoungsoo Jung, Akbar Sharifi, Praveen Yedlapalli, Vinay Saripalli, Adwait Jog, Nachiappan Chidambaram, Sai Prashanth Muralidhara, Shekhar Srikantaiah, Tao Zhang, Anirban Chatterjee, Manu Shantharam, Shankar Prasad Sastry, and Nicholas Voshell.

I was lucky to have many friends who continuously supported me throughout my PhD. They were with me in the toughest, most stressful times of my life, and their friendship enabled me to get through. I would like to express my thanks to my dearest friends Mahmut Aktasoğlu, Şafakcan Tunçdemir, Sena, Dağştan, and Defne Şahin, Umut Orhan, Onur Kayıran, Taha Özdemir, İdil Akçetin, and Ayşe Özgür Pehlivan.

And my family... My dear mother and father were always there for me. They never forced me into doing anything, nor were they ever over-protective of me. They always made me feel their support which was all I needed. And my brother; he has always been my role model. I simply do not have the words to explain my gratitude towards them.
To the memory of my grandfather Mehmet Akgül...
Chapter 1

Introduction

*High performance computing, or supercomputing* is a form of computing that uses the cutting edge of technology. High performance computing systems, also known as supercomputers, are vital for solving complex problems which either cannot be solved or take impractical amount of time to solve using other systems. Supercomputing is used for making advances in many technical fields that result in pioneering improvements in the welfare of humanity. By using supercomputers, scientists gather information about and shed light onto various phenomena; researchers perform simulations that to replace experiments that are dangerous, expensive, or impossible to conduct; and engineers quickly evaluate various design choices to reduce the time to market and improve the quality of products. Applications of supercomputing include, but are not limited to intelligence, defense, climate prediction, plasma physics, computational biology, societal health and safety, earthquake analysis, geoscience, astrophysics materials science, and computational nanotechnology. The advances in the state of the art in these fields are closely tied to the advances in supercomputing.
1.1 Definition of a Supercomputer

A supercomputer is a complete computing system that delivers the highest performance achievable by the state-of-the-art computing technology. This definition includes the hardware, the systems software (operating system and compiler/linker), and the application to be executed on the system. The term supercomputing comprises various activities related to the design, manufacturing, operation, and use of supercomputers.

1.2 Supercomputer Architecture

Supercomputer hardware consist of compute units, memory, I/O system, and an interconnect. The compute units can be in the form of general purpose processors, specialized graphics processors (GPUs), application-specific integrated circuits (ASICs), field programmable gate arrays (FPGAs), or any combination of these; and they are responsible for executing the operations in the application program running on the supercomputer. This execution comprises performing arithmetic and logical calculations, initiating memory accesses, and controlling the flow of program execution. The memory system stores the current state of computa-
Figure 1.2. A blade of Titan contains 4 nodes that communicate with other blades using Cray’s proprietary Gemini [1] interconnect (the two large chips on the left). When combined, the four nodes on a blade contain four AMD Opteron 6274 [2] processors with 16 cores each (the four large chips arranged vertically at the center), 32GB DDR3 memory (16 vertical DIMMs on the left), four Nvidia Tesla K20X GPUs [3] (four chips/heat sinks on the right), 24GB GDDR5 memory to be used by the GPUs (two vertical DIMMs at the center), and four chips that convert hypertransport interface of Opteron CPUs to PCI-E interface of Tesla GPUs (four small gray heat sinks on the right-middle).

1.3 Types of Supercomputers

There can be various ways to classify supercomputers, but the classification adopted in this work is based on the degree at which the supercomputer is specialized for
a given application or class of applications. On one end are application-specific supercomputers that consist of completely custom designed ASIC compute units and custom interconnect. On the other end are general purpose supercomputers that typically contain commodity workstations or servers and use off-the-shelf processors and networking.

Studies indicate that power considerations will force system architects to rely more on application-tailored designs in the near future [13]. An application-specific supercomputer uses compute units that have been specialized for one particular application, such as a custom ASIC designed for scientific computing. In case of scientific computing, the interconnect is also typically specialized to provide high bandwidth for both memory accesses and inter-processor communication. Custom ASICs can have customized on-chip storage that can provide much higher bandwidth than any general purpose processor. They can also employ better latency hiding mechanisms, examples of which include better pipelining of operations and multi-threading. Because they are designed for a specific class of applications, these systems are not manufactured in high volumes, and therefore are relatively expensive. Further, they typically use less advanced semiconductor technology than commodity processors, typically lagging by a one or two of process technology generations (e.g., 22 nm vs 32/45 nm). Consequently, they typically have lower clock frequencies and sequential performance, which means that they can outperform general purpose systems only if they can realize much higher amount of parallelism. Yet, even on a scalable application, there can be serial parts or parts that do not scale very well, which may not be suitable for parallel execution, and lead to performance degradation due to this frequency disadvantage. Therefore, it is also typical for application-specific systems to also contain one or more general purpose processors for running these sequential parts of the application
without any penalties. In this case, the application-specific hardware is considered as an *application-specific hardware accelerator* that is used for improving the performance of parallel parts of the application.

An application-specific supercomputer can also use field programmable gate arrays (FPGAs) in place of ASICs. In comparison with ASICs, FPGAs have much lower cost and can realize faster time to market, but they have less on-chip resources, and therefore, lower compute power. On the other hand, the flexibility of reprogramming renders FPGAs very convenient for rapid prototyping and testing of real systems. One example of a system that uses many FPGAs together to solve a complex problem is the RAMP system [4], shown in Figure 1.3. The RAMP system uses multiple BEE3 boards [5] to conduct research on the design, evaluation, and scalability testing of futuristic multiprocessor systems. Each BEE3 board, shown in detail in Figure 1.4, is custom designed to have four FPGAs as well as various high bandwidth interconnects to realize communication across chips on the same board as well as across chips on different boards.

A general purpose supercomputer, on the other hand, is built using off-the-shelf processors developed for workstations or commercial servers connected by an off-the-shelf network using the I/O interface of the processor. Such machines are often referred to as clusters because they are constructed by grouping workstations or servers. Although the efficiency of these systems (e.g., the computational capability per processor or performance per watt) is typically much less than what can be achieved by an application-specific system, building a supercomputer using general purpose computers significantly reduces the acquisition cost of the system and is a versatile solution that can be easily reprogrammed to solve multiple problems.

Clearly, there are supercomputers that do not fit either definition and fall in between these two categories. For instance, a supercomputer can be general pur-
Figure 1.3. The RAMP [4] multi-FPGA reconfigurable system designed for emulation of multiprocessors systems.

pose and be used for any type of application, while also having special features for accelerating the execution of a specific class of applications. In fact, when building supercomputers for scientific applications, supercomputing companies already make several critical optimizations in the design. For instance, the Titan system from Cray is a product for scientific computing and therefore is designed to have higher memory per node, with custom interconnection chips for higher inter-node communication bandwidth, and a tight integration between CPUs and GPGPUs to increase floating point arithmetic throughput.

1.4 Increasing Parallelism for Higher Performance

The foremost and most widely adopted way to improve compute power of a system is to perform multiple computations simultaneously: parallelism. All modern computers use some sort of parallelism, examples of which include pipelining that enables multiple computations to overlap, multi-threading that switches one thread that encountered a long latency operation with another thread to hide the latency
Figure 1.4. Organization of a BEE3 board [5] that is used in the RAMP system. Each board has four Virtex-5 FPGAs connected using a ring interconnect, 4 GbE and 8 CX4 connections for communicating across boards, and up to 64GB of DDR2 memory on 16 DIMMs. 

of the operation, and non-blocking processors/caches/memories that can continue executing or receiving more requests while serving a cache miss or a memory access. Being at the cutting edge of technology, supercomputers take parallelism to the limits achievable by the current technology. Relying especially on thread-level parallelism, supercomputers employ massive amounts of parallelism by using thousands of threads and processors. These processors, running simultaneously towards solving a single problem, tremendously increase the computational capability (i.e., the ability to solve a single problem) of the supercomputer\(^1\) enabling the solution of problems of size or complexity that no other computer can solve.

While some problems can be easily decomposed into subproblems that are distributed to be executed on different processors to be solved independently with little or no overheads (also known as embarrassingly parallel applications), there are lots of problems are not so easy to decompose or distribute. The most common cause for this behavior is dependencies among the parallel subproblems that

---

\(^1\)Another metric for supercomputer performance is computational capacity, which identifies the ability of the supercomputer in solving a large number small, independent problems.
require frequent communication of data between processors. On a supercomputer system, this communication is done using memory: one processor writes data to memory, and the other one reads it. As a result, these frequently communicating applications put great pressure on the memory system, rendering the optimization of the memory system in a supercomputer system a very critical task.

1.5 When Parallelism is Not Enough: The Memory Problem

Various reports on new high performance computing challenges indicate that future’s exascale machines will suffer from various memory-related problems such as less memory bandwidth per core, higher relative memory access latencies, and relatively energy-inefficient memory accesses [14, 15, 13, 16]. Technology trends show that processor performance and memory bandwidth are diverging, posing a major challenge for computer architects. As processor performance increases, increasing memory bandwidth to maintain a constant ratio becomes prohibitively costly, and in fact in the long run, becomes impossible due to limits of interconnect scaling. The evolution of DRAM row access latency is also not able to keep up with the continuing increase in processor performance, resulting in a relative increase in DRAM latency when expressed in terms of instructions processed while waiting for a DRAM access. In addition to memory bandwidth and latency problems, memory power consumption is also becoming a very serious issue. The per-bit energy consumption of memory systems are unlikely to see significant improvements, making it very difficult to meet the energy challenges posed by exascale computing.

Considering the massive amount of parallelism realized by supercomputers,
this discrepancy between processor performance and memory performance makes the memory the primary bottleneck in supercomputers. If the memory system is unable to provide the data to the compute units at a fast enough rate due to bandwidth and latency issues, we cannot keep these compute units busy and suffer from underutilization. However, if we scale the memory systems at the same rate as we scale processors to meet those performance demands, we will experience skyrocketing power consumption due to energy-inefficient memory systems.

The next two sections examine how the scalability problem of memory impacts application-specific hardware accelerators and general purpose supercomputer systems individually.

1.5.1 The Memory Problem in Application Specific Systems

On the application-specific hardware accelerators side, memory is an important factor that limits the scalability of the entire system. In an application-specific hardware accelerator that is automatically synthesized from a sequential software representation of an application, a shared memory that is accessed by all compute units cannot be scaled up to a large design as this would require a prohibitively large network that connects the compute units and memory. Instead, distributed memory systems where each node has a fast local memory is preferred, yet there are many challenges in realizing scalable designs. Progress in automatic partitioning of data in compilers has been limited and the preferred way of generating distributed memory systems has been manual partitioning by the programmer. This not only is a hard task, but also leads to suboptimal results as it is very difficult, if not impossible, to manually identify the optimum data partitioning.
Further, using distributed systems does not solve the memory scalability problem of hardware accelerators either, as the bottleneck now becomes the maintenance of coherence across the distributed memories. This coherence problem especially restricts application-specific hardware, because the most important benefit of using ASICs, namely, the ability to have custom on-chip storage that can deliver higher bandwidth at lower latency, takes a big hit. Keeping all memories coherent also means that all on-chip caches must be kept coherent, no matter whether they reside on the same chip or different chips, the overheads of which can easily negate all benefits from using custom hardware.

These memory trends are likely to lead to new hardware developments, examples of which include memories with limited coherence guarantees and specialized memory systems [14, 15]. If these advancements can solve the coherence problem in application-specific hardware accelerators, then using custom on-chip cache hierarchies with simplified coherence hardware can provide orders of magnitude improvements in energy efficiency as on-chip memories (and lower level caches in cache hierarchies) can easily achieve much better bandwidth-per-watt values than off-chip main memory (and higher level caches) [17].

1.5.2 The Memory Problem in General Purpose Systems

On the general purpose supercomputers side, increasing intra- and inter-chip parallelism keeps increasing the peak performance that can be delivered, but the average performance values are typically much lower. A chief reason for this inefficiency is poor memory performance. Technology trends show that memory latencies are unlikely to improve substantially [13]. Further, modern many-core processors [18, 2, 3] employ complex on-chip interconnects that connect a large number of cores and
on-chip caches, which results in a high end-to-end network latency that must also be suffered by the memory operations. Using bandwidth optimized graphics memory, which is a design choice that prioritizes bandwidth over latency, is yet another factor that increases memory access latency. Large memory access latencies can be difficult to hide, which can lead to underutilization of the compute units in the processors. A successful latency-hiding mechanism is required to sustain high performance. Such mechanism would allow the processor to initiate many memory references before the response to the first reference is received, thereby allowing the processor to fill in a memory system pipeline.

Despite all optimizations towards improving memory bandwidth, memory systems still fail to match the increasing demand from the processors due to the rapid increase in the number of cores integrated on-chip. While accessing a local, private on-chip cache can provide a very high bandwidth to the corresponding local core, the off-chip memory bandwidth is shared by all the cores in the system and making memory bandwidth a very scarce resource in many-core systems. Memory bandwidth is expected to become an even more serious problem in the future due to the disparity in the scaling of processor core counts and memory interface bandwidth. Therefore, we need methods to better utilize memory bandwidth, such as methods to save redundant off-chip memory traffic.

Memory also poses significant energy efficiency challenges for general purpose systems. Today's DDR-3 memory interface technology consumes about 70 pJ/bit, and the projections to DDR-5 in year 2018 show that it is expected to reduce to 30 pJ/bit [13]. Within the same time period, a 5-10X improvement in the energy efficiency of floating point arithmetic is expected. Clearly, the poor energy efficiency scaling of memory will also become a significant problem for general purpose systems.
1.6 Compiler-based Optimizations to Address Memory Problems

While there has been a huge body of work in optimizing compilers for parallel systems [19, 20], there are still no absolute solutions to the memory problems identified above for the two types of high performance computing systems. Specifically,

- To address the scalable distributed memory design problem of application-specific accelerators, we need an automatic synthesis tool that can generate an application-specific hardware accelerator with a memory system that can scale to hundreds or thousands of ASIC or FPGA chips. Chapter 2 of this dissertation gives the details of this scalability problem in application-specific hardware accelerators. A compiler method to solve this problem is proposed and a compiler infrastructure that employs this method to address this problem is presented. The proposed method generates a multi-level memory hierarchy in application-specific hardware accelerators that eliminates the need for using coherence hardware across memories, thereby resulting in a scalable memory system.

- To improve the performance and efficiency of general purpose supercomputers, we need effective memory latency hiding and memory bandwidth saving techniques. One such latency hiding mechanism is prefetching which initiates a request for a data item before the actual instruction that requires the data is ready to be executed. Prefetching overlaps memory accesses with the execution of other instructions, effectively hiding the latency of the memory accesses. Special streaming store instructions can also be used towards saving precious memory bandwidth. A regular store to a memory location that does not exist in the on-chip cache requires a read from off-chip memory, an update, and then (at the time of cache line eviction) a write-back to the off-chip memory, to ensure cache coherence. On the
other hand, by identifying stores that show streaming behavior, we can replace these stores with special store instructions that can eliminate the redundant read from memory. Software prefetching for latency hiding and using specialized store instructions for bandwidth saving have not been explored on emerging processors with many cores and wide vector units. Chapter 3 of this dissertation proposes compiler optimizations that target a recently released, state-of-the-art many-core system with wide vector units (i.e., the Intel Xeon Phi coprocessor). Results show that these compiler-based methods are very effective in hiding memory access latencies and reducing the memory bandwidth requirements of applications.
Automatic Synthesis of Scalable Memory Systems for Application-Specific Hardware Accelerators

The key difficulties in designing memory hierarchies for future computing systems with extreme scale parallelism include (i) overcoming the design complexity of system-wide memory coherence, (ii) achieving low power, and (iii) achieving fast access times within such a memory hierarchy. Towards addressing these difficulties, this chapter proposes 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, this 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 without 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. This chapter also provides 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.

2.1 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 [21, 22] 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 [23, 24] 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.

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 dif-
fers 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 [25]. 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 [26, 27] 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.

The remainder of this article is organized as follows. In Section 2.2, we first de-
scribe our target architecture, namely, application-specific hardware accelerators, and then give an overview of our compiler/profiler toolchain. Section 2.3 summarizes the related work in the area and underlines the novel parts of this study. Section 2.4 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 Section 2.5. 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 Section 2.6 and conclude with Section 2.7.

2.2 Overview of the Proposed System

2.2.1 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 [28, 29]. 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 [30], 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 a loop to an inner loop state machine by simply treating the loop as a multi-cycle pipelined operation. Duplciating 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 a 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 [31, 24]. Partitioned memory is synthesized 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 can automatically create a multi-level memory hierarchy to realize a scalable memory system. Figure 2.1 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
Figure 2.1. 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 Figure 2.4.

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 Figure 2.1 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 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.

2.2.2 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\(^1\) generated by the gcc 4.5.4 compiler with the “-gstabs+” option (“-gstabs+” adds 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

---

\(^1\)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 [32, 33, 34] which allows access to a wide range of applications.
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. 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 [35], 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: (i) 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 (ii) making \( I_1, I_2 \) depen-
(2) there can be some runtime information needed to identify that a dependence is not observed in practice for 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.

**Compiler/Profiler Toolchain:** Viewed at a high level, our compiler/profiler toolchain operates in three stages (as shown in Figure 2.2):

- **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

dent 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, 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.
analysis results with the dynamic dependence analysis results from Profile-1 (if available) (i) to verify the correctness of compiler dependence analysis tests, and (ii) 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 system and obtain information about the usage patterns of the memories extracted by the compiler.

### 2.3 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.
2.3.1 Compiler-Based 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 [36, 37], and therefore, has better memory performance.

Another use of memory partitioning is to improve the performance of software managed, on-chip scratch pad memories [38, 39]. 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 [40] shift the memory partitioning task to the programmer. In PGAS languages [41, 42, 43, 44], 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 difficult 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, 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 [45]. 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, 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

$^3$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 $\text{for}(\text{int } i=0; i<n\text{Proc}; ++i) \text{func}(i, \text{data}[i])$; created by a typical sequential to parallel code transformation [36] 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 identical application-specific hardware units implementing $\text{func}$, each with its private child memory hierarchy representing a partition of the $\text{data}$ array, without using any coherence hardware across child memory hierarchies, and without using power-hungry general purpose processors to implement $\text{func}$. Our techniques, however, are not limited to scientific applications.
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 [23] 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 [24] 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. [46, 47] 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 translation of such a shared medium into low-overhead point-to-point links. So et al. [48] 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 [49], Cong et al. [31] 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. Instead, our primary

\[^4\]While the present work is a limit study, it is instructive to consider its practical implementa-
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, precisely address these issues not addressed by prior work.

### 2.3.2 Symbolic Execution in Compilers

Symbolic execution [50] has been used widely in various contexts [51, 52, 53, 54, 55], yet its use in optimizing compilers has been limited [25]. We identified two main deficiencies in its use in the prior work and devised solutions.

---

Scroll down to see the next page.
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” symbolic expression, “contents of address” symbolic expression) pairs (as opposed to traditional [25] (“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 Section 2.5.1).

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 [56]. 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 [25]), 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.

### 2.3.3 Profile-Based Memory Partitioning

Profiling has been used in program analysis for various purposes, examples of which include identifying hot regions for mapping to accelerators [57], disambiguating dependences for speculative optimizations [58], and identifying parallel loops [59, 60]. 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. [61] 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.

### 2.4 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.

2.4.1 Accelerator Memory Model

As explained in Section 2.2, 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 $A$.

2.4.2 Partitioning the Memory of a Procedure

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

\(^5\)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.

\(^6\)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., $max(a, b)$).
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\textsuperscript{8} to find out if any two given memory instructions are dependent.

The pseudocode for our memory partitioning algorithm is given in Figure 2.3. In this algorithm, we first construct an undirected dependence graph $(V, E)$, where

\textsuperscript{8}We provide a definition for dependence in Section 2.5.
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
15     // Each component is a new memory
16     mpart = empty memory partition;
17     mpart.setID(getUniqueMemoryID());
18     // Instructions in the component are connected to the new memory
19     for i ∈ Memory Instructions of comp do
20         mpart.add(i);
21     end
22     PartitionTable[i] = mpart;
23     memories.add(mpart);
24 end
Figure 2.3. The pseudocode for our memory partitioning algorithm.

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 [62] 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 [63] to find the connected components of this
graph, which has a complexity of $O(|V| + |E| \times \alpha(|V|))$, where $\alpha$ is the inverse Ackermann function.

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.
2.4.3 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 Figure 2.4. 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
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

Figure 2.4. Hierarchical memory partitioning example. The parallel hardware for this code was depicted in Figure 2.1. 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 \( \log_2(M) \) 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 Section 2.4.4.

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
input: List of memory instructions in target program fragment, program CFG
output: Hierarchical Memory Partition Table,
        Memory Hierarchy Tree, Generated Memories Table
ScopeList = find all program scopes of CFG for memory partitioning;
SortedScopeList = sort ScopeList using entry BB reverse post order numbers;
// Initialize all scopes to have empty set of generated memories
foreach Scope S ∈ SortedScopeList do
    GeneratedMemories[S] = empty set of memories;
end
MEM0 = getUniqueMemoryID(); // Memory hierarchy tree is initialized to have a single root node
MemoryHierarchyTree = MEM0;
// All memory instructions are initially connected to MEM0
foreach Instruction i ∈ CFG.getInstructions() do
    HierarchicalPartitionTable[i] = MEM0;
end
// Partition memories at each program scope sequentially
foreach Scope S ∈ SortedScopeList do
    // Get instructions in scope S
    (ChildPartitionTable, ChildMemories) = partitionMemoryAtScope(S);
    foreach instruction-memory pair (i, childMEM) ∈ ChildPartitionTable do
        // Get the old memory for i
        parentMEM = HierarchicalPartitionTable[i];
        // Update with child memory in the hierarchical partition table
        HierarchicalPartitionTable[i] = childMEM;
        if childMEM ∉ MemoryHierarchyTree then
            // Save the (parentMEM, childMEM) pair to the memory hierarchy
            MemoryHierarchyTree.addNode(childMEM);
            MemoryHierarchyTree.addEdge((parentMEM, childMEM));
        end
    end
    // Record the memories generated at this scope
    foreach memory childMEM in ChildMemories do
        GeneratedMemories[scope].add(childMEM);
    end
end

Figure 2.5. The pseudocode for our hierarchical memory partitioning algorithm.

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 Figure 2.5, 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. 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.

2.4.4 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, 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 Figure 2.6 (and also in Figure 2.4). 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 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, $MEM_a$ and $MEM_b$. This would violate our assumption where $MEM_a$ and $MEM_b$ 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, in a customized application-specific environment, the dirty
and valid bits can be reduced or eliminated at compile time. For example, when a memory is accessed only as 32-bit words, only one dirty/valid bit pair is needed every 32-bits.

**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 [31] 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 Section 2.4.2). 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.
2.5 Program Analysis for Memory Partitioning

As described in Section 2.4, 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.

2.5.1 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 [64]. The accuracy at which these symbolic address expressions can be disambiguated determines the final quality of memory parti-

\(^9\)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.
tioning. 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\(^\text{10}\) known to hold when an actual program execution reaches this particular program point.

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 \(\ast (\&y)\) and \(\ast (\&z)\)) must be 0 and 1, respectively.

**Iterative Symbolic Execution:** Symbolic execution of a procedure (or any part of a procedure), given in Figure 2.7, works as follows. We execute a pass over the entire input procedure by symbolically executing the instructions in all basic blocks in the CFG\(^\text{11}\). This operation starts with an empty program state (i.e., a program state with no facts and *true* path predicate) corresponding to the start

\(^{10}\)Existing symbolic execution methods [25] define a program fact as a mapping from variable names to their values, such as \((\text{var}, \text{val})\) mapping variable \(\text{var}\) to its value \(\text{val}\). We modified this definition to record program facts involving variables without names and pointers, such as \((\text{ptr}, \text{val})\) which maps the location pointed by \(\text{ptr}\) to \(\text{val}\). We can still capture variable to value mapping information, albeit at a slightly modified form, such as \((\&\text{var}, \text{val})\), where \(\&\text{var}\) is the address expression for variable \(\text{var}\).

\(^{11}\)The basic blocks are traversed based on their reverse post ordering numbers.
Figure 2.7. The pseudocode for our symbolic execution algorithm. Please see the appendix for more details of the subroutines `execute` and `join`.

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

Figure 2.8 presents the pseudocode for the `execute` function referred to in Figure 2.7. 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
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
4 // arithmetic operation
5 case ADD:
6 Expr e0 = state.lookup(i.src[0]);
7 Expr e1 = state.lookup(i.src[1]);
8 Expr result = simplify(ADD, e0, e1);
9 state.update(i.dst[0], result);
10 break;
11 end
12 // memory load operation
13 case LOAD:
14 Expr e0 = state.lookup(i.src[0]);
15 Expr result = state.lookup(e0);
16 state.update(i.dst[0], result);
17 break;
18 end
19 // branch on condition register
20 case BR:
21 Expr cond = state.lookup(i.src[0]);
22 Expr condn = simplify(NOT, cond);
23 b.exitStates[0].facts = state.facts;
24 b.exitStates[1].facts = state.facts;
25 Expr p1 = simplify(AND, state.pred, cond);
26 Expr p2 = simplify(AND, state.pred, condn);
27 b.exitStates[0].pred = p1;
28 b.exitStates[1].pred = p2;
29 break;
30 end
31 // Cases for other operations
32 ...
33 endsw
34 end

Figure 2.8. Symbolic execution execute algorithm for executing the instructions in a basic block.

more exit program states. 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
Figure 2.9. Symbolic execution `join` algorithm for joining the program states corresponding to two incoming forward CFG edges to a basic block.

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.

Figure 2.9 gives the pseudocode for the `join` function referred in Figure 2.7. 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))

```plaintext
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
outState.pred = simplify(OR, inState1.pred, inState2.pred);

// Add facts from first edge
for fact f1 ∈ inState1.facts do
  Expr val1 = f1.value;
  Expr val2 = inState2.lookup(f1.address);
  Expr val = simplify(f1.pred ? val1 : val2);
  outState.addFact(f1.address, val);
end

// Add facts from second edge
for fact f2 ∈ inState2.facts do
  // Handled in first loop?
  if f2.address exists in inState1.facts then
    continue;
  end
  Expr val1 = inState1.lookup(f2.address);
  Expr val2 = f2.value;
  Expr val = simplify(f2.pred ? val2 : val1);
  outState.addFact(f2.address, val);
end
```

```
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.\textsuperscript{12} 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 \texttt{execute} function given in Figure 2.8. 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 Figure 2.7.

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.\textsuperscript{13} 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

\textsuperscript{12} If the values on both states are the same, the expression simplification algorithm eliminates the conditional expression, i.e., $\texttt{simplify}(\texttt{cond} ? \ x : \ x) = x$.

\textsuperscript{13} 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.
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 \((\text{addr, iv})\) to the entry edge program state of the corresponding loop, where \(\text{addr}\) represents the address expression of the induction variable and \(\text{iv}\) 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., \(ivs\)) [65], we use pattern matching. If an induction variable matches a known 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.\(^{14}\) 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.\(^{15}\)

**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).

\(^{14}\)Solving induction variables of inner loops before outer loops enables us to solve higher order induction variables.

\(^{15}\)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 \(0 \leq I < I_{max}\)).
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 $\text{cond} \ ? \ (A + \text{offset}) : (B + \text{offset})$, which means that it refers to array $A$ or $B$ based on the value of variable $\text{cond}$ 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\footnote{\textsuperscript{16}In our implementation, we use the symbol information embedded into the assembly files by \texttt{gcc} when the \texttt{gstabs+} option is used.} 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 $\texttt{int}$ and $q$ is a pointer to $\texttt{float}$.

5. We also apply other standard dependence tests such as the range test \cite{66} and the GCD test \cite{67}. 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 procedures cannot be aliased, the user can help the compiler perform better symbolic program analysis and better memory disambiguation.

2.5.2 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. We generate two versions of the target procedure: an instrumented version and an original non-instrumented version. We execute the profiled version 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 with 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 (i) have never been observed to depend on each other at runtime, (ii) are not proved to be independent by static analysis, (iii) are speculatively linked to different memories, and (iv) 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.

2.6 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.

2.6.1 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:

\[
\begin{align*}
    u(I_1, I_2, R) &= \begin{cases} 
        1, & \text{if } d_c(I_1, I_2, R) = \text{may-alias and } d_p(I_1, I_2, R) = \text{no-alias} \\
        0, & \text{otherwise}
    \end{cases} 
\end{align*}
\] (2.1)

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 (i) answered by the compiler-based dependence analysis as *may-alias* and (ii) answered by the profiler-based dependence analysis as *no-alias*. The mathematical formula for static accuracy is as follows.

\[
StaticAccuracy(R) = 1 - \frac{\sum_{I_1} \sum_{I_2 > I_1} u(I_1, I_2, R)}{\sum_{I_1} \sum_{I_2 > I_1} 1} \leq \frac{(N - 1)}{N/2}.
\] (2.2)

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.

\[
DynamicAccuracy(R) = 1 - \frac{\sum_{I_1} \sum_{I_2 > I_1} u(I_1, I_2, R) \cdot \text{Weight}(I_1, I_2, R)}{\sum_{I_1} \sum_{I_2 > I_1} \text{Weight}(I_1, I_2, R)},
\] (2.4)

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:

$$Weight(I_1, I_2, R) = \max(count(I_1, R), count(I_2, R)). \quad (2.5)$$

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 [68], defined as follows.

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

$$Address\ Bits(X) = \log_2(|X|) \quad (2.6)$$

**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:

$$Entropy(X) = -\sum_{i=1}^{n} p(x_i) \cdot \log_2(p(x_i)) \quad (2.7)$$

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 2.10 gives a proof of concept for entropy encoding of memory addresses over an example. Huffman encoding [69] 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 Figure 2.10(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 Figure 2.10(b). 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 Figure 2.10(c). 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
Figure 2.10. 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).

WE1-WE3 signals at the instant WE0 becomes high (active) at an early time, the decoder circuit could use, e.g., CMOS domino logic [70]. 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 nearly zero. 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 work.

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 $y_i$:

$$
\text{EffectiveNumberOfMemories}(Y) = 2^{-\sum_{i=1}^{n\text{memories}} p_m(y_i) \log_2(p_m(y_i))}
$$

(2.8)

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.

### 2.6.2 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 [6] 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 Figure 2.11(a) [6]. 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. Figure 2.11(b) 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 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 2.12(a) shows the relationship between the number of index bits in a cache (where cache capacity is proportional to \(2^{\text{addr.bits}}\)) 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 Figures 2.12(b) and 2.12(c), 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...
Figure 2.12. (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 $2^{addr_bits}$.

into physical metrics (i.e., latency and energy) related to the memory hierarchy.
Table 2.1. Benchmarks used in evaluation and their hottest functions targeted by our compiler for memory partitioning. The last two columns give the static and dynamic accuracy of compiler-based dependence analysis.

### 2.6.3 Benchmarks

In our evaluation, we use benchmarks from the SPEC CPU 2000 and CPU 2006 benchmark suites [26, 27]. For each benchmark, we identified the hottest procedure shown in Table 2.1 and inlined all calls to user functions within that procedure. We are using a set of nine 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.

### 2.6.4 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 Section 2.6.1, we evaluated 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 dependence questions posed in the compiler. In case of dynamic accuracy, 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 2.1 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.

### 2.6.5 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 Section 2.5. We then compare these two sets of results and discuss the underlying reasons for any similarities and differences we observe. However, the results presented in this document are obtained only for one level of partitioning (for the outermost program scope).

**Compiler-Based (conservative) Memory Partitioning Results.** Table 2.2 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
Table 2.2. Compiler-based (conservative) memory partitioning results: average address entropy and average number of address bits without partitioning and with partitioning. 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).

<table>
<thead>
<tr>
<th></th>
<th>w/o Partitioning</th>
<th>w/ Partitioning</th>
<th>Memories (Scalar/Array)</th>
<th>Effective Num. of Memories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Entropy</td>
<td>Bits</td>
<td>Entropy</td>
<td>Bits</td>
</tr>
<tr>
<td>art</td>
<td>16.4</td>
<td>16.6</td>
<td>13.6</td>
<td>13.7</td>
</tr>
<tr>
<td>equake</td>
<td>16.8</td>
<td>15.9</td>
<td>7.5</td>
<td>7.8</td>
</tr>
<tr>
<td>bzip2</td>
<td>9.3</td>
<td>16.6</td>
<td>5.3</td>
<td>5.3</td>
</tr>
<tr>
<td>gcc</td>
<td>5.0</td>
<td>5.5</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>mcf</td>
<td>11.9</td>
<td>12.3</td>
<td>9.2</td>
<td>9.4</td>
</tr>
<tr>
<td>milc</td>
<td>6.2</td>
<td>6.9</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>sjeng</td>
<td>6.2</td>
<td>6.5</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>lbm</td>
<td>9.3</td>
<td>15.2</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>astar</td>
<td>6.4</td>
<td>10.9</td>
<td>3.0</td>
<td>4.9</td>
</tr>
</tbody>
</table>

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 2.2), 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
entropies 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 falling 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 2.13 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 2.2 “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 Figure 2.2). 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.
Table 2.3. Profiler-based (optimistic) memory partitioning results: average address entropy and average number of address bits without partitioning and with partitioning. 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).

<table>
<thead>
<tr>
<th>Bench.</th>
<th>w/o Partitioning</th>
<th>w/ Partitioning</th>
<th>Memories (Scalar/Array)</th>
<th>Effective Num. of Memories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Entropy</td>
<td>Bits</td>
<td>Entropy</td>
<td>Bits</td>
</tr>
<tr>
<td>art</td>
<td>15.9</td>
<td>16.6</td>
<td>14.5</td>
<td>14.5</td>
</tr>
<tr>
<td>quake</td>
<td>10.8</td>
<td>15.9</td>
<td>5.9</td>
<td>6.1</td>
</tr>
<tr>
<td>bzip2</td>
<td>9.3</td>
<td>16.6</td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td>gcc</td>
<td>5.0</td>
<td>5.4</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>mcf</td>
<td>11.9</td>
<td>12.3</td>
<td>8.8</td>
<td>8.9</td>
</tr>
<tr>
<td>milc</td>
<td>6.2</td>
<td>7.0</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>sjeng</td>
<td>6.1</td>
<td>6.4</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>lbm</td>
<td>9.6</td>
<td>15.3</td>
<td>3.3</td>
<td>3.3</td>
</tr>
<tr>
<td>astar</td>
<td>6.4</td>
<td>10.7</td>
<td>1.8</td>
<td>1.8</td>
</tr>
</tbody>
</table>

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 2.3, when compared to the equivalent results with compiler-based partitioning given in Table 2.2, 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, 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 2.14 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.

**Discussion of Results.** While the results presented in Tables 2.2, 2.3 and Figures 2.13, 2.14 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 2.1 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 $I_1$ and $I_2$ in the code fragment shown in Figure 2.15(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 outperform-
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);
  ...
}

Figure 2.15. 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 [i1start, i1end] and [i2start, i2end] do not overlap in practice. The profiler, on the other hand, can identify that this dependence never occurs at runtime.

In sjeng, the compiler-based partitioning. Finally, in bzip2, we observe that the profiler is in fact extracting information that is impossible for the compiler to obtain at compile time. We repeat a simplified version of this code fragment in Figure 2.15(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.
2.7 Conclusion

This chapter 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 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. Therefore, we believe that automatic memory partitioning can be effective in solving the memory scalability problem, enabling the realization of custom, application-specific memories.
Latency Hiding and Bandwidth
Saving Memory Optimizations for
General Purpose Many-core
Processors

The Intel Xeon Phi coprocessor based on the Intel Many Integrated Core Architecture has software prefetching instructions to hide memory latencies and special store instructions to save bandwidth on streaming non-temporal store operations. In this chapter, we provide details on compiler-based generation of these instructions and evaluate their impact on the performance of the Intel Xeon Phi coprocessor using a wide range of parallel applications with different characteristics. Our results show that the Intel Composer XE 2013 compiler can make effective use of these mechanisms to achieve significant performance improvements.
3.1 Introduction

The Intel Xeon Phi coprocessor based on the Intel Many Integrated Core Architecture (Intel MIC Architecture) is a many-core processor with long vector (SIMD) units targeted for highly parallel workloads in the High Performance Computing (HPC) segment. It is designed to provide immense throughput: a single chip can deliver a peak performance of well above one double precision TeraFLOPS. High performance computing systems can significantly increase their computational capabilities by accommodating many Intel Xeon Phi coprocessors and distributing the workloads across these coprocessors [71].

However, achieving performance close to the peak performance value of the Intel Xeon Phi coprocessor requires its cores and vector units to be kept busy at all times, which in turn requires data to be delivered at a very high rate without stalls. Yet, providing sustained low memory latency is more challenging with the Intel MIC architecture than the Intel Xeon processors. This is mainly because (i) as opposed to traditional large, power-hungry, out-of-order cores which are known to be good at hiding the latency of cache/memory accesses by exploiting instruction level parallelism, Intel Xeon Phi coprocessor uses small, power efficient, in-order cores, (ii) the last-level cache capacity per core (and per thread) is smaller on the Intel Xeon Phi coprocessor than the latest Intel Xeon processors, and (iii) Intel Xeon Phi coprocessor has a higher memory access latency than Intel Xeon processors.

Intel Xeon Phi coprocessor relies on software and hardware data prefetching techniques to bring data into local caches ahead of need to hide the latency of accessing caches and memory. It also employs special vector instructions that can be used for streaming stores in order to save memory bandwidth. While those
prefetch and streaming store instructions can be inserted by expert programmers using intrinsics, in practice, it is a hard and error-prone task to identify when these instructions will be useful and with which parameters they should be used to bring benefits. It is much more practical to reduce the role of the programmer to a high level control over these optimizations and delegate the compiler to be responsible for performing the bulk of the work: analyzing the source code, identifying the cases where these instructions can be used, and generating them. In this chapter [72],

- We present how the Intel Xeon Phi coprocessor software prefetch and streaming store instructions are generated by the Intel Composer XE 2013 compiler, and
- We evaluate the impact of these mechanisms on the overall performance of the Intel Xeon Phi coprocessor using a variety of parallel applications with different characteristics.

Our experimental results demonstrate that (i) a large number of applications benefit significantly from software prefetching instructions (on top of hardware prefetching) that are generated automatically by the compiler for the Intel Xeon Phi coprocessor, (ii) some benchmarks can further improve when compiler options that control prefetching behavior are used (e.g., to enable indirect prefetching), and (iii) many applications benefit from compiler generated streaming store instructions.

The remainder of this chapter is organized as follows. In Section 3.2, we provide the details of the Intel Xeon Phi coprocessor based on the Intel MIC architecture. We present the software/hardware data prefetching mechanisms and streaming store instructions in this architecture together with how they are generated automatically by our compiler to improve application performance in Sections 3.3 and 3.4. Section 3.5 gives the details of our experimental framework and presents an in-depth analysis of the impact of data prefetching and streaming stores on
our target benchmarks. We give the related work in Section 3.6 and conclude the chapter with Section 3.7.

3.2 Background

Intel Xeon Phi coprocessor is designed to serve the needs of applications in the HPC segment that are highly parallel, make extensive use of vector operations, and are occasionally memory bandwidth bound. We start this section with a presentation of the salient architectural characteristics of this coprocessor, contrasting it with the Intel Xeon processor E5-2600 series product family. We then approach to this new architecture from a programmer’s point of view. Before moving on to the details of compiler-based data prefetching and streaming store instructions in the next sections, we also give an overview of our compiler.

3.2.1 Architectural Overview of the Intel Xeon Phi Coprocessor

Cores and Parallelism. Targeting highly parallel applications, the Intel Xeon Phi coprocessor [73], the organization of which is shown in Figure 3.1, employs 60 (or more) cores, each core supporting four hardware thread contexts which gives a total of 240 (or more) threads that can run simultaneously. The cores, the details of which are shown in Figure 3.2, are in-order and can issue two instructions per cycle. Two instruction-issue is coupled with two pipelines (u and v-pipes), which improves the utilization of the units inside the core. The two pipes are not symmetrical and there are restrictions on the type of instructions can be scheduled on each pipe (e.g., vector operations can only be scheduled on the u-pipe). At
each cycle, the instruction decoder picks instructions from the threads scheduled on the core in a round-robin order with the restriction that at least one cycle is needed between decoding instructions from the same thread. Hence, if only one thread is scheduled on a core, its instructions can be issued at every other cycle. This restriction is a design choice to balance the latency of the pipeline stages in the processor and prevent the decode stage from reducing the clock frequency.

Overall, in contrast to the Intel Xeon processors, the Intel Xeon Phi coprocessor trades off core complexity for coarse-grain parallelism. While the former family employs up to eight complex out-of-order, four-issue cores, the Intel MIC architecture makes use of a larger number of simpler in-order cores running at a lower frequency. These cores are instances of the Intel P54C architecture that is extended based on the needs of target applications (e.g., 64-bit addressing to satisfy the need for using large memory, long vector units to improve performance of data-parallel workloads). The reduction in the complexity of the cores enables significant reduction in the core silicon area and power consumption, which in turn enables the integration of a larger number of cores on chip. Despite its sacrifice from single-thread performance and instruction level parallelism, Intel Xeon Phi products can realize significantly higher thread level parallelism.

**Vector (SIMD) Operations.** To improve the performance of applications that make extensive use of vector operations, the Intel Xeon Phi coprocessor employs 512-bit vector units. Using a 512-bit vector unit, 16 single precision (or 8 double precision) floating point (FP) operations can be performed as a single vector operation; and with the help of the fused multiply-add (FMA) instruction, up to 32 FP operations can be performed at each core at each cycle.

In comparison to the current 128-bit SSE and 256-bit AVX vector extensions, this new coprocessor can pack up to 8x and 4x the number of operations into a
single instruction, respectively.\footnote{Current SSE/AVX instructions do not include FMA. Intel announced that FMA will be available in its next microarchitecture in 2013. AMD started supporting FMA instructions with its recent architecture.}

**Caches and Main Memory.** Each Intel Xeon Phi coprocessor core has a private 512KB L2 cache that is kept coherent using a distributed tag directory. The caches, directories, memory controllers, and the PCIe interface are connected using two unidirectional ring networks. A miss in a local L2 cache (that can be a hit in a remote L2 cache or trigger an off-chip memory access) must travel through this interconnect to reach its destination. Having more blocks connected to the ring network increases the average number of hops each message has to travel on the network, which in turn increases the total access latency.
Figure 3.2. Details of a single core in the Xeon Phi Coprocessor, taken from [7].

While the Intel Xeon Phi coprocessor has a total on-chip cache capacity of 30MB which is higher than the sum of L2 and L3 caches on Intel Xeon processor E5-2600 product family, its per-core cache capacity is actually smaller. Compared to 256KB L2 and 2.5MB L3 effective cache capacity per-core, an Intel Xeon Phi coprocessor core has only 512KB L2 cache. Further, this space is shared by four threads. This reduction in per core cache capacity is a design choice that enables such a high number of cores to be integrated in a single chip. As a result, the on-chip cache in this architecture is more limited and effective management of its capacity is more important to achieve high performance.

The Intel Xeon Phi coprocessor resides on a separate add-in card connected to a host system through PCIe. The memory system on the card is completely independent from the host memory. The card has 16 channels of GDDR5 memory that are controlled by 8 on-chip memory controllers. Operating at a 32-bit interface and 5.5GT/s speed, the memory subsystem has a maximum memory bandwidth of 352GB/s. On the other hand, Intel Xeon processor E5-2600 product family members have 4 channels that can deliver up to 51.2GB/s maximum bandwidth.
per socket at DDR3-1600. With its bandwidth advantage, Intel Xeon Phi products can be expected to achieve higher performance on applications that are memory bandwidth bound.

Although the Intel Xeon Phi coprocessor has a high memory bandwidth, it also has a high off-chip memory access latency, mainly because the GDDR5 memory it uses is optimized for bandwidth rather than latency, and therefore has a higher access latency than the DDR3 memory technology [74] used with other Intel Xeon processors.

Considering the smaller per-core cache capacity, an increase in off-chip memory access latency, together with its in-order cores that are not able to hide cache/memory access latency as out-of-order cores, optimizing the cache/memory behavior of applications becomes critical in reaching high performance with Intel Xeon Phi products. In an unoptimized case, application threads will frequently miss in local L2 caches and stall for extended periods of time until the needed data is retrieved from remote L2 or off-chip memory, which will under-utilize the core computational resources and lead to poor performance.

In Sections 3 and 4, we show two important methods to alleviate these issues: data prefetching and streaming store instructions. Our work demonstrates compiler-based generation of these instructions and evaluates their impact on performance.

### 3.2.2 Programming and Optimizing for the Intel Xeon Phi Coprocessors

After providing an architectural overview of the Intel Xeon Phi coprocessor, we now provide a view of this coprocessor from a programmer’s angle. For further
details on programming for this architecture, we refer the reader to [75, 76].

**Usage Models.** There are two usage models for the Intel Xeon Phi coprocessor: offload and native. In this work, we evaluate applications that make use of either type of usage model. For our applications that use the offload usage model, we start the target application on the host processor and offload scalable parallel regions of the application to the coprocessor. This requires transferring data from the host processor to the coprocessor upon entry to offload regions and in the other direction upon exit from offload regions. For our applications that use the native usage model, we compile the entire application for the coprocessor. We start the application directly on the coprocessor and execute only on it. In this usage, application data is allocated only on the coprocessor and is never transferred between the host processor and the coprocessor. Both models can be used to structure MPI programs that scale to multiple processors and coprocessors. In the native model, processors and coprocessors both have MPI ranks, whereas in the offload model, only the processors have MPI ranks and each processor can use its coprocessors to offload its work.

**Optimizations.** There are fundamental analyses and optimizations that must be performed to achieve high performance from the Intel MIC architecture such as parallelization (i.e., scalability analysis), vectorization, code transformations, data layout transformations, etc. The most important property an application must possess in order to be suitable for the Intel MIC architecture is scalability: it must scale well to hundreds of threads to effectively utilize the large number of cores. However, it must be kept in mind that the highest performing configuration is not always the one that uses the highest number of threads (or even cores). For instance, while maximum performance can be obtained by scattering one thread per core for one application, fully utilizing all hardware thread contexts and using a
balanced thread mapping can be better for another. The number of threads/cores used and the affinity setting chosen can significantly affect performance. Using more threads per core can better hide cache/memory access latencies through fine-grain hardware multithreading, but it also reduces the effective cache/TLB capacity and bandwidth per thread.

As the Intel MIC architecture also relies on effective utilization of its long vector units to achieve high performance, a significant portion of the application should be vectorizable and the vectorized code should efficiently use all the lanes in the vector units. There can be various reasons for underutilization of vector units, the most important ones being non-unit-stride and unaligned memory accesses. For instance, using arrays of structures can lead to constant, but non-unit stride accesses. In this case, the programmer can manually perform AOS to structure of arrays (SOA) data layout transformations to improve performance.

The Intel MIC architecture vector units have a vector length of 512-bits (64B). A vector load/store that is aligned to 64B boundary can be performed as a single instruction, whereas an unaligned vector load/store requires two separate memory accesses (\texttt{vloadunpacklo/vpackstorelo} and \texttt{vloadunpackhi/vpackstorehi}) and a merge/split operation (performed implicitly) on them to write/read the data to/from a vector register. To obtain the benefits of aligned memory instructions, one must (i) instruct the compiler and runtime system to allocate data items from aligned memory locations and (ii) inform the compiler that it is safe for it to generate aligned memory instructions when accessing those data items.

There are also various other code transformations that can significantly impact the performance of applications on the Intel MIC architecture. For instance, loop tiling cache locality optimization can make a bigger impact due to its higher off-chip access latency, and loop unrolling can lead to better core utilization due to
having simpler in-order cores.

While we briefly mentioned these optimizations in this section for completeness, these topics are not the main subject of this study. We refer the interested reader to various articles published in the literature for multi-core and vector architectures. In this study, we work with applications that are highly threaded, efficiently vectorized, and optimized by the programmer and/or the compiler for the Intel MIC architecture.

3.2.3 Intel Composer XE 2013

Intel Composer XE 2013 is an optimizing C/C++/Fortran compiler that delivers high performance for applications running on Intel Core or Intel Xeon processors, Intel Xeon Phi coprocessors, and IA-compatible processors. It includes state-of-the-art compiler optimizations including but not limited to code restructuring, inter-procedural optimizations, memory optimizations, partial redundancy elimination, and partial dead store elimination. The compiler performs automatic parallelization and automatic vectorization. It also supports various directive-based methods to express parallelism and vectorization. Parallel execution can be obtained using OpenMP pragmas or Intel Cilk Plus task/data parallelism constructs. The programmer can manually annotate loops that needs to be vectorized using the \texttt{#pragma simd} pragma, or use the Intel Cilk Plus array notation that helps the compiler in generating vector code. For complex loops with function calls that cannot be vectorized even after function inlining, elemental functions can help in obtaining vector code. Further details about the compiler can be found in [77].
3.3 Prefetching on the Intel Xeon Phi Coprocessor

The Intel Xeon Phi coprocessor relies heavily on prefetching for hiding cache and memory access latency. It has two prefetching mechanisms. The first is software prefetching that uses explicit prefetch instructions. The second is hardware prefetching that is transparent to software, which dynamically identifies cache miss patterns and generates prefetch requests. Both methods aim to retrieve data ahead of need to prevent blocking of application threads due to cache misses.

3.3.1 Software Prefetching

Software prefetching involves static identification of memory accesses in a program and insertion of prefetch instructions for these memory accesses such that the prefetched data will be readily available in the on-chip caches when the memory access is executed. While prefetch instructions can be inserted by expert programmers using low-level intrinsics, this is a tedious and error-prone task. The compiler incorporates advanced techniques to do efficient software prefetching for memory accesses.

Architectural Support. The Intel MIC architecture contains two types of software prefetching instructions. The \textit{vprefetch1} instruction brings a 64B line of data from memory to L2 cache, and \textit{vprefetch0} further pulls it to the L1 cache. The cache line allocated for the prefetched data is put into shared (S) coherence state in the MESI protocol. There is also another variant of both prefetch instructions that mark the cache line as exclusive (E) in the tag directory (\textit{vprefetch1} and \textit{vprefetch0}) [73]. Prefetching loads data into the cache speculatively, which may
lead to attempting to access memory locations that are not allocated to the current
process. In this case, the hardware does not generate page faults and silently
ignores the prefetch instruction. This eliminates the need to use guarding bound-
check statements around prefetch instructions.

**Prefetch Distance Computation.** Loops constitute almost all of the execution
time of applications and typically access large ranges of memory regions, and
therefore, they are the best targets to apply data prefetching. As temporal locality
in L1 cache and register promotion can simply hide the latency of scalar memory
accesses, the main concern for a prefetcher is aggregate variables. In a loop, hiding
the latency of an access to an array location that will be used in one loop iteration
typically requires it to be prefetched a number of loop iterations before that iter-
ation. In other words, these prefetches target array locations that will be accessed
in the future iterations of the loop. Prefetch distance is defined as how many loop
iterations into the future the generated prefetch instructions target. Since the
compiler performs prefetching after vectorization, prefetch distances are expressed
in terms of vector loop iterations. The compiler can also generate prefetches for
vector load/store intrinsics inserted by the programmer, and pointer accesses that
are similar to array accesses, where target address can be predicted in advance.
It supports address calculations that involve affine functions of surrounding loop
indices and functions of those indices expressed using additional instructions in the
loop (that can be identified by the compiler using induction variable substitution).

Given a loop with a memory access to associate a prefetch with, there are two
measures that determine the optimum distance for the prefetch. The first one
is the expected latency of the prefetch operation, i.e., how long it will take the
prefetch to complete. The second one is the rate of execution of the loop, i.e., how
rapidly will the program execution progress and reach the actual memory access.
Ideally, the prefetch operation should complete just before the memory access is encountered, otherwise either the latency will not be hidden completely or a useful block may be evicted from the cache.

The factors that determine the latency of a prefetch operation are (i) the latency of accessing the target level in the memory hierarchy (more than ten cycles to access local L2 cache, hundreds of cycles to access main memory), and (ii) the amount of load in the system that adds additional queuing delay on top of these base latency values.

Calculating the rate of execution of a loop involves finding the expected total latency of executing one iteration of the loop. This can be especially complicated (i) when there are inner loops, and (ii) there are multiple threads sharing the instruction issue cycles. In the presence of inner loops, we use inner loop trip count estimation to identify the latency of executing each inner loop (also including the impact of prefetching for those inner loops).

While deciding on prefetch distances and generating prefetches, the compiler also takes into account the impact of issuing those prefetches on the TLB pressure during execution. If the distance chosen for a prefetch is likely to cause pressure on the TLB, then the compiler reduces the prefetch distance to make sure no TLB thrashing occurs due to prefetching. The -opt-report compiler option gives detailed information on the software prefetches inserted by the compiler and the distances used for them.

**Outer Loop Prefetching.** For data accesses in inner loops of a loop nest, the compiler decides whether to prefetch for the future iterations of the inner loop or the outer loop. This decision is based on heuristics such as the estimated trip counts of the loops and symbolic contiguity analysis of data accesses in the inner loop. Clearly, it is important for the compiler to make accurate trip count estimates to
make the right decision. The compiler not only easily identifies constant trip count loops, but also uses various other methods to identify trip counts such as analyzing array dimensions. The programmer can also assist the compiler by providing loop trip count hints (e.g., \texttt{#pragma loop count(200)}).

**Initial Value Prefetching.** Prefetching for future loop iterations does not eliminate cache misses for the first few iterations of loops. To handle these misses, the compiler inserts initial value prefetches before loops. This optimization is enabled by default in the compiler and is especially useful for loops with small trip counts.

**Exclusive Prefetching.** If a data that is being prefetched will immediately be written by the prefetching thread, then we use exclusive prefetch instructions to prefetch it. This saves an extra access to the tag directory during the subsequent store instruction to change the state of the cache line from shared to exclusive. While exclusive prefetching can improve performance in this case, if accidentally used too aggressively, it can cause invalidation of useful data in the caches of other cores and lead to ping-pong of data across cores. Therefore, the compiler generates exclusive prefetches only when it is confident that the cache line will be updated and is not shared across threads.

**Indirect Access Prefetching.** Indirect array accesses do not follow simple strided access patterns and two consecutive indices may access data that are far apart and on different cache lines in the target array. A typical example is when an index array is used to access a second array, as given by the \texttt{A[B[i]]} notation. Indirect accesses are frequently observed in scientific applications, such as those working with sparse matrices that are stored and accessed using the indices of the non-zero elements for space efficiency. When the target array is large, accesses to it become cache misses and the memory latency becomes the main factor that
limits performance. Performance can be improved by issuing prefetches to hide the latency of these indirect memory accesses. At loop iteration $i$, if we want to prefetch for $D_1$ iterations ahead (i.e., prefetching $A[B[i+D_1]]$), we are also making a demand access to $B[i+D_1]$. Therefore, to make sure the access to the index array (i.e., $B[i+D_1]$) will also become a cache hit, we need to make another prefetch for some further $D_2$ iterations (i.e., prefetch $B[i+D_1+D_2]$).

The Intel Xeon Phi coprocessor packs up to 16 iterations of a scalar loop into one vector iteration. Our compiler generates gather/scatter instructions ($vgather/vscatter$) when it vectorizes loops with indirect array accesses. A gather/scatter instruction reads/writes all the elements in the vector register from/to memory sequentially, one by one. As a result, to prefetch for an indirect gather/scatter instruction, our compiler generates up to 16 prefetch requests to cover all locations that will be accessed by the gather/scatter instruction. Meanwhile, the index array is accessed sequentially and only one prefetch operation is sufficient to bring 16 index values into the cache.

**Controlling the Compiler Prefetching Behavior.** While our compiler can perform all the associated analyses and generate software prefetches fully automatically, it also supports programmer control of various prefetching parameters through compiler options and pragmas for fine tuning. Compiler options modify the default behavior for an entire compilation unit and pragmas operate at a finer granularity and override the compiler options for their target loops or functions. The compiler supports the following mechanisms to alter the default behavior of the software prefetcher (for prefetching to L2 cache, L1 cache, or both):

- Disable software prefetcher (option and pragma): The programmer may choose to disable generation of software prefetch instructions. For instance, if the hardware is able to prefetch equally well in an application, then we may be
able to save some cycles spent for fetching/issuing/scheduling software prefetch instructions by disabling software prefetching. Another example is an advanced programmer who disables compiler-based data-prefetching to manually insert prefetch instructions using intrinsics.

- Set prefetch distance (option and pragma): The programmer may choose to control the prefetch distance manually. This could be useful when the compiler is not able to make a good decision on the prefetch distance, for instance when there is an inner loop with an unknown or variable trip count.

- Disable exclusive prefetch generation (option): While the default behavior of the compiler is to generate exclusive prefetches for store instructions, this behavior can be controlled using a compiler option.

- Enable indirect prefetch generation (option): In its default behavior, the compiler does not generate indirect prefetches. Indirect prefetching can be enabled by using a compiler option.

### 3.3.2 Hardware Prefetching

The Intel Xeon Phi coprocessor employs a 16-stream hardware prefetcher [73] that is enabled by default when the system starts. It is trained as it detects L2 cache misses and, upon detection of a cache miss pattern, it starts issuing prefetch requests to memory. While we believe there is no reason for any user to turn off the hardware prefetcher, we also experiment when hardware prefetcher is disabled in order to report the isolated performance of our compiler-based software prefetching and analyze the interaction of the hardware prefetcher with software prefetching.
3.3.3 Combined Software/Hardware Prefetching

When hardware and software prefetching are used together, a cache miss successfully converted into a cache hit by the software prefetcher will not train the hardware prefetcher. Therefore, for those streams of accesses, the hardware prefetcher will not generate any prefetch requests. On the other hand, if there are some streams that are not handled by the software prefetcher, the hardware prefetcher may still be able to detect them at runtime and issue prefetches for them. One example is when the compiler is unable to analyze an array subscript expression and cannot generate any prefetches. In this case, the hardware prefetcher may be able to detect a memory stream and generate prefetches at runtime.

3.4 Streaming Store Instructions on the Intel Xeon Phi Coprocessor

Streaming store instructions are specialized memory store instructions designed to save off-chip memory bandwidth in cases where data is being streamed into memory. Unlike regular stores, streaming store instructions do not perform a read-for-ownership (RFO) for the target cache line before the actual store. The rationale behind this is that any data read from memory for this purpose will anyways not be used and will get overwritten by the data stream.

In this section, we provide the details of the streaming store instructions on the Intel Xeon Phi coprocessor, the conditions under which our compiler generates those instructions, and software-controlled cache line eviction which is closely associated with memory accesses.
3.4.1 Types of Streaming Store Instructions

The streaming store instructions on the Intel Xeon Phi Coprocessor are vector instructions that operate on data whose length is equal to the cache line size. There are only unmasked, aligned versions of these instructions. Therefore, they can only be used when the target store address is aligned to 64B cache line size and the store operation is unmasked (i.e., the entire vector will be written, not a part of it). If the store is unaligned or masked, regular store instructions (\textit{vpackstore} or \textit{vmov}) must be used. Streaming store instructions simply allocate a cache line and then set its entire contents. There are two types of streaming store instructions on the Intel Xeon Phi coprocessor: NR and NR.NGO stores. Both of them can be used to save bandwidth, but their performance and applicability are different.

**NR Stores.** The NR store instruction (\textit{vmovnr}) is a standard vector store instruction that can always be used safely. An NR store instruction that misses in the local cache causes all potential copies of the cache line in remote caches to be invalidated, the cache line to be allocated (but not initialized) at the local cache in exclusive state, and the write-data in the instruction to be written to the cache line. There is no data transfer from main memory which is what saves memory bandwidth. An NR store instruction and other load and/or store instructions from the same thread are globally ordered, which means that all observers of this sequence of instructions always see the same fixed execution order.

**NR.NGO Stores.** The NR.NGO (non-globally ordered) store instruction (\textit{vmovnrngo}) relaxes the global ordering constraint of the NR store instruction. This relaxation makes the NR.NGO instruction have a lower latency than the NR instruction, which can be used to achieve higher performance in streaming store intensive applications. However, removing this restriction means that an NR.NGO
store instruction and other load and/or store instructions from the same thread can be observed by two observers to have two different orderings. The use of NR.NGO store instructions is safe only when reordering the order of these instructions is verified not to change the outcome. Otherwise, using NR.NGO stores may lead to incorrect execution. Our compiler can generate NR.NGO store instructions for store instructions that it identifies to have non-temporal behavior. For instance, a parallel loop that is detected to be non-temporal by our compiler can make use of NR.NGO instructions. At the end of such a loop, to ensure all outstanding non-globally ordered stores are completed and all threads have a consistent view of memory, our compiler generates a fence (a lock instruction) after the loop. This fence is needed before continuing execution of the subsequent code fragment to ensure all threads have exactly the same view of memory.

It is important to make a comparison of the streaming store instructions on the Intel Xeon Phi coprocessor and other Intel processors. The SSE instruction set extensions first introduced the movntpd instruction that stores a full 16B SSE vector register to memory. There is only a packed variant of this SSE instruction, which means that the target address must be 16B aligned in memory. Note that while the main memory interface operates at a 64B cache line granularity, the size of data being written using this instruction is only one fourth of a cache line. A typical implementation uses an on-chip buffer to merge the streaming stores that fall into the same cache line and write them together to off-chip memory as a single operation. As Intel Xeon Phi coprocessor vector length is four times that of SSE and is equal to the cache line size, it does not need a separate buffer to combine multiple stores. It is also important to note that streaming stores on the Intel MIC architecture are cached, whereas the implementation on the Intel Xeon processors makes them bypass the on-chip cache hierarchy and get combined on a separate
buffer.

3.4.2 Cache Line Eviction Instructions

Any memory load or store operation that targets a memory location that will not be reused in the immediate future is non-temporal. These non-temporal memory accesses do not benefit from caching and letting them occupy space in the cache until they become the least recently used entry can be a waste of those cache lines. Evicting those cache lines can free up cache space that can be used for caching other data lines. Intel Xeon Phi has special instructions for evicting cache lines from L1 cache and from L2 cache (`clevict0` and `clevict1`). The compiler generates these instructions on loops that are identified to show non-temporal behavior. Note that generating cache line eviction instructions for loops is rather easy on the Intel Xeon Phi coprocessor, because its vector length is equal to the cache line size, which means that one cache line can be evicted per vector loop iteration.

3.4.3 Controlling the Compiler Streaming Store and Cache Line Eviction Behavior

For a non-temporal loop, the default behavior of the compiler is as follows:

- For aligned stores: Generate NR.NGO store instructions, do not generate any prefetch instructions, and generate cache line eviction instructions after NR.NGO stores,

- For unaligned stores: Generate packed store instruction pairs (i.e., two instructions for each store that write the high and low halves of the vector register to cache with mask), generate prefetch instructions, generate cache line eviction instructions to evict the cache lines that are fully written,
• For loads: Generate appropriate load instructions based on alignment (i.e., 
\textit{vloadunpacklo/hi} or \textit{vmov}), generate prefetch instructions, do not generate cache 
line eviction instructions.

The compiler has heuristics to identify non-temporal loops fully automatically.
When these heuristics identify a loop to be non-temporal, the default behavior of 
the compiler for streaming stores is to generate NR.NGO stores and a fence (a 
\textit{lock} instruction) after the loop (to ensure safety). However, these fully automatic 
heuristics kick-in rarely because they need to know that the trip count of the 
target loop is large. The programmer can assist the compiler in identifying non-
temporal loops by manually marking loops as non-temporal using the \texttt{#pragma 
nontemporal} pragma or by using the \texttt{-opt-streaming-stores always} compiler option 
which declares all loops in the target compilation unit as non-temporal. Then, the 
compiler generates NR.NGO instructions for these non-temporal loops and does 
not generate fences after them. In this case, the programmer is responsible for 
making sure that this use of NR.NGO instructions is safe.

Alternatively, the programmer can instruct the compiler to generate safe NR 
stores instead of NR.NGO stores using a compiler option, in which case, fences 
are not needed. The compiler also has an option for the programmer to manually 
disable the generation of the cache line eviction instructions.

If an NR or NR.NGO streaming instruction is not used for a store operation 
that shows streaming characteristics (e.g., because it is unaligned), the compiler 
generates prefetches for this instruction. While prefetching will hide the latency 
of accessing off-chip memory to bring the original cache line contents, it will not 
have the same memory bandwidth saving benefit of the NR and NR.NGO store 
instructions. When these special store instructions are used, the compiler does 
not generate any software prefetch instructions for them (generating prefetches for
them will negate their bandwidth-saving effects), and since these instructions do not generate any L2 cache misses, they do not train the hardware prefetcher either.

3.5 Evaluation and Results

We now present the details of our evaluation for the performance impact of generating prefetch and streaming store instructions on applications executing on Intel Xeon Phi. We use an experimental version of the Intel Composer XE 2013 to perform the experiments in this paper.

3.5.1 Target Applications

In our evaluation, we use applications from various sources. We use 8 benchmarks from [78] which provided a detailed evaluation and analysis of these benchmarks from vectorization and parallelization perspectives. We use *swim* and *mgrid* benchmarks from the SPEC OMP 2001 suite with OMPL ref inputs [79]. *swim* is a scalable benchmark, whereas *mgrid* is a moderately scalable benchmark. We report the total (end-to-end) performance of these two benchmarks as well as the isolated performance of their most important three parallel loops.² We use the *Stream* benchmark [80] that is widely used for memory bandwidth testing. *CG* from NAS Parallel Benchmarks [81] is used with its class-B inputs to demonstrate the impact of indirect prefetching. We also use *Leukocyte*, which is a highly parallel benchmark from Rodinia Benchmark Suite [82], and an in-house kernel that performs one dimensional FFT. These applications are executed in native model on Intel Xeon Phi coprocessor, which means that the entire application runs on

²When these applications are executed in parallel on our Intel Xeon processor, these top three parallel loops constitute 90% of the execution time for *swim* and 83% of the execution time for *mgrid*. 
the coprocessor card and it makes no data transfer with the host system. We also provide results with two in-house kernels (3DFD and TifaMMMy) that run in offload model whose sequential parts are run on the host machine and parallel kernels are run on Intel Xeon Phi coprocessor. A list of our target applications is provided in Table 3.1. All of our target applications are parallelized using OpenMP pragmas and are vectorized by the compiler.

### 3.5.2 Target System Specifications

The details of the Intel Xeon Phi coprocessor we used in this work to evaluate the effectiveness of data prefetching and streaming store instructions are provided in Table 3.2. Although it is not our primary goal, we also provide some performance comparison with a state-of-the-art Intel Xeon processor based 2-socket server. The details of this system, which is also used as the host system for applications executed in offload model, are also listed in Table 3.2.

<table>
<thead>
<tr>
<th>Source</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel [78]</td>
<td>cv (2D 5x5 conv.)</td>
</tr>
<tr>
<td></td>
<td>bp (backprojection)</td>
</tr>
<tr>
<td></td>
<td>lbc (libor)</td>
</tr>
<tr>
<td></td>
<td>nb (nbody)</td>
</tr>
<tr>
<td></td>
<td>rd (radar)</td>
</tr>
<tr>
<td></td>
<td>tp (TPCF)</td>
</tr>
<tr>
<td></td>
<td>ts (treesearch)</td>
</tr>
<tr>
<td></td>
<td>vo (vortex)</td>
</tr>
<tr>
<td>SPEC OMP 2001 [79]</td>
<td>sw (swim)</td>
</tr>
<tr>
<td></td>
<td>mg (mgrid)</td>
</tr>
<tr>
<td>In-house kernels</td>
<td>3d-offload (3DFD)</td>
</tr>
<tr>
<td></td>
<td>ti-offload (TifaMMMy)</td>
</tr>
<tr>
<td></td>
<td>ld (1DFFT)</td>
</tr>
<tr>
<td>Stream [80]</td>
<td>st (Stream)</td>
</tr>
<tr>
<td>NPB [81]</td>
<td>cg (Conjugate Gradient)</td>
</tr>
<tr>
<td>Rodinia [82]</td>
<td>lc (Leukocyte)</td>
</tr>
</tbody>
</table>

Table 3.1. Target parallel applications.
### Table 3.2. Target system specifications.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Xeon Phi</th>
<th>Xeon E5-2687W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chips</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Cores and Threads</td>
<td>61 and 244</td>
<td>8 and 16 per socket</td>
</tr>
<tr>
<td>Frequency</td>
<td>1 GHz</td>
<td>3.092 MHz</td>
</tr>
<tr>
<td>Data Caches</td>
<td>32 KB L1, 512 KB L2 per core</td>
<td>64 KB L1, 256 KB L2 per core, 20 MB L3 per socket</td>
</tr>
<tr>
<td>Power Budget</td>
<td>300 W</td>
<td>150 W per socket</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>7936 MB</td>
<td>16 GB per socket</td>
</tr>
<tr>
<td>Memory Technology</td>
<td>GDDR5</td>
<td>DDR3</td>
</tr>
<tr>
<td>Memory Speed</td>
<td>2.75 GHz (5.5 GT/s)</td>
<td>667 MHz (1333 MT/s)</td>
</tr>
<tr>
<td>Memory Channels</td>
<td>16</td>
<td>4 per socket</td>
</tr>
<tr>
<td>Memory Data Width</td>
<td>32 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>Peak Memory Bandwidth</td>
<td>352 GB/s</td>
<td>41.6 GB/s per socket</td>
</tr>
<tr>
<td>Vector Length</td>
<td>512 bits</td>
<td>256 bits (AVX)</td>
</tr>
</tbody>
</table>

#### 3.5.3 Results

In our first set of experiments, we analyze the impact of software and hardware prefetching on Intel Xeon Phi coprocessor when the generation of streaming stores is disabled in our compiler. Then, we use the best performing prefetcher configuration as our baseline and evaluate the impact of enabling streaming store generation in our compiler.

**Prefetching Analysis.** To analyze the impact of hardware and software prefetching on Intel Xeon Phi coprocessor for our target benchmarks, we performed prefetching experiments with the following five configurations: (i) HWP: hardware prefetching only, (ii) NOP: no hardware and no software prefetching, (iii) SWP: default software prefetching only, (iv) HWP+SWP: hardware prefetcher enabled, default software prefetching, (v) HWP+SWP_OPT: hardware prefetcher enabled, manually tuned software prefetching. Based on our results, we categorize the target benchmarks as software prefetching sensitive and insensitive.

Figure 3.3 shows our results with benchmarks that benefit from software prefetching by at least 3%. The performance numbers on the y-axis are normalized with
Figure 3.3. Results with benchmarks that benefit from software prefetching.

Figure 3.4. Results with benchmarks that are insensitive to software prefetching.

respect to using the hardware prefetche alone (HWP). For these benchmarks, we typically see that the third bar (SWP) is higher than the first bar (HWP), which indicates that software prefetching performs better than hardware prefetching for them. For most of these benchmarks, there is no difference between SWP and HWP+SWP values, which means that enabling hardware prefetcher when we already have software prefetching does not improve performance. Comparing the HWP+SWP and HWP+SWP_OPT bars, we see that ti-offload, st-triad, sw-loop1, and cg gained from manually tuning software prefetching. In ti-offload, we disabled MEM-to-L2 software prefetching and let hardware prefetcher carry out this task. In st-triad and sw-loop1, we increased the default prefetch distance to 64 and 8 iterations for MEM-to-L2 and L2-to-L1 prefetching, respectively. In cg, we manually enabled indirect prefetching in the compiler, which further improved the performance of this benchmark by about 50%. For these benchmarks, the gains from software prefetching range from 5% (cv) to 81% (cg).
Our results with benchmarks that are insensitive to software prefetching are shown in Figure 3.4. There can be two reasons for an application to be insensitive to software prefetching. First, the hardware prefetcher may already be doing a good job in converting many L2 cache misses into hits. Second, the application may have good cache locality (e.g., a good reuse pattern or a cache-resident data set) which translates into an already very low cache miss rate. As a result, we further classify our target benchmarks as hardware prefetching sensitive and insensitive. For the benchmarks on the left hand side of Figure 3.4, we see that the bars 1 and 5 (HWP and HWP+SWP_OPT) are almost equal, which means that software prefetching does not improve their performance. On the other hand, the differences between bars 1 and 2 (HWP and NOP) are very large. This means that the hardware prefetcher plays a vital role for these benchmarks. The performance degradation from disabling hardware prefetching in these benchmarks range from 11% (ts) to 87% (sw-loop3).

Most of the benchmarks on the right hand side of the same figure (bp, lbc, rd) are from [78]. These benchmarks are not sensitive to any kind of prefetching because they either use small amount of memory that fit into the on-chip cache or have their code optimized (e.g., using tiling) for the cache capacity of Intel Xeon Phi coprocessor. The st-copy benchmark is also not sensitive to any prefetching because the compiler automatically converts the copy loop into a memcpy operation. This memcpy operation is an optimized library function that implicitly uses software prefetch instructions that we cannot control using the compiler. As a result, in the NOP configuration where we disable the hardware prefetcher and software prefetch generation in the compiler, it continues to perform good as it still benefits from software prefetching. For this set of benchmarks, disabling the hardware prefetcher leads to less than 2% performance degradation.
Streaming Store and Cache Line Eviction Analysis. Our next study is to identify how much performance benefits Intel Xeon Phi coprocessor streaming store and cache line eviction instructions can bring on top of optimized prefetching. For this purpose, we used the optimized prefetching configuration (HWP+SWP_OPT) from our prefetching analysis above as our baseline, and generated four configurations. Each configuration uses either NR or NR.NGO stores (shown in the figures as +NR or +NGO). Two out of four configurations also have cache line eviction enabled (shown as +CLE). For this analysis, we add a compiler option (-opt-streaming-stores always) that marks all loops in the benchmarks as non-temporal loops. As a result of this option, benchmarks that actually do not contain loops with non-temporal behavior are also forced to be compiled assuming they are non-temporal. As we will see with the results, this not only helps us identify which benchmarks/loops can benefit from streaming stores and cache line eviction, but also shows us how much performance degradation can occur if we incorrectly mark loops as non-temporal.

Note that there are only aligned versions for the streaming store instructions. Therefore, the compiler can only generate these instructions if a store in a non-temporal loop is aligned. When there are multiple unaligned array accesses in a loop, the compiler uses loop peeling to align one of these array accesses to the 64B boundary. The compiler further uses multiversioning, where it generates two versions of the loop. In one version, a second array access is also performed using aligned memory instructions, whereas the other version uses aligned instructions only for the array access that is aligned using loop peeling. The decisions of which array accesses will be aligned using loop peeling and multiversioning are based on heuristics in the compiler. The heuristics prioritize arrays that are accessed more than once in the loop body where aligning one access automatically aligns
other accesses as well (e.g., aligning $A[i]$ also aligns $A[i+64]$). When a loop is either identified by the compiler or marked by the programmer to be non-temporal, then the store instructions in the loop are prioritized to be aligned using loop peeling and multiversioning to enable generation of more efficient NR/NR.NGO store instructions.

We again classify our target benchmarks, this time we group them based on whether they benefit from the use of streaming store instructions. Figure 3.5 shows our results with benchmarks that receive more than 3% performance improvement when using some combination of streaming stores and cache line eviction. Clearly, the biggest improvements are obtained with the tests $st\text{-}scale$, $st\text{-}add$, and $st\text{-}triad$. In these tests, using NR.NGO stores brings 21-45% performance improvement. This is an expected result as $Stream$ is a benchmark that is designed to be bandwidth bound. By eliminating redundant reads from memory using streaming store instructions, a higher $Stream$ score is obtained. $sw\text{-}loop1$ and $sw\text{-}loop2$ also have significant 12% performance improvement, which is the primary reason for having 7% performance improvement in $sw\text{-}total$. While the benefit on $cg$ is also high (13%), the remaining three benchmarks show relatively small but steady improvements in the 3-5% range. Enabling cache line eviction while also having some form of streaming stores improved the performance in only $mg\text{-}loop1$. For this loop, the +NGO+CLE configuration achieved 4% speedup.

For most of the remaining benchmarks, shown in Figure 3.6, there was either no impact on performance or a degradation less than 10% was observed. However, there were a couple of benchmarks that had significant performance degradations. For $lbc$ and $ti\text{-}offload$, marking all loops non-temporal and using cache line eviction for all store instructions caused 3-10X penalty in performance. The reason for this degradation was the elimination of cache locality over the outer loop in a loop.
nest. When a vectorized inner loop in a loop nest uses a small output array that
is defined over an outer scope (e.g., the body of the outer loop), then the small
trip count inner loop can reuse these arrays from the cache for consecutive outer
loop iterations. However, when the programmer forces the compiler to treat the
inner loop as non-temporal, the compiler evicts these arrays from the cache at the
inner loop. Then, in the next iteration of the outer loop, accesses to these arrays
become cache misses. These benchmarks show that using cache line eviction in-
structions for temporal loops can severely degrade performance. Therefore, when
evaluating the impact of streaming store generation by adding the -opt-streaming-
stores always compiler option, it is recommended for programmers to also make
experiments with cache line eviction turned off. Using the compiler option marks
all loops in the target compilation unit as non-temporal and a performance degra-
dation on one loop due to cache line eviction can hide all the benefits obtained
on other loops. After carefully analyzing the impact of streaming store and cache
line eviction instructions, for applications that degrade by using these instructions,
the programmer should not force the compiler to generate these instructions by
removing any non-temporal pragmas or options. This would be equivalent to using
the best prefetching configuration.

Comparison with Intel Xeon Processor based System. We also eval-
uated a subset of our benchmarks on our dual-socket Intel Xeon processor based
system to compare its performance with the Intel Xeon Phi coprocessor. For this
analysis, we used Stream, swim, and mgrid benchmarks, and our results are pre-
sented in Figure 3.7. The 2.2-2.3X difference between the two systems in Stream is
the result of the bandwidth on Intel Xeon Phi coprocessor being much higher.\footnote{The peak st-triad value we obtained in our tests with Intel Xeon Phi based system was 157GB/sec. While this is much lower than the 352GB/sec maximum memory bandwidth, st-triad performance on this system is actually limited by the performance of the on-chip interconnect}
the other two applications, we see that Intel Xeon Phi coprocessor performs about 1.7X better for swim, but performs similar to Intel Xeon processor based system for mgrid. The reason for this behavior is that swim is a scalable benchmark that is very suitable for this many-core architecture, whereas mgrid is a moderately scalable benchmark. For the rest of the benchmarks (from Intel[78]) in Figure 3.7, Xeon Phi coprocessor achieved performance improvements in the range of 1.2X (ts) to 4.8X (lbc), with the geometric mean improvement being 2.1X. This is an expected outcome as these benchmarks were specifically written and optimized to efficiently exploit the Intel MIC architecture.

that transfers data between the on-chip caches and the memory controllers.
3.6 Related Work

Software and Hardware Prefetching. Starting with the first proposals for compilers generating non-blocking prefetch instructions to improve cache performance [83, 84] and using specialized hardware for data prefetching [85, 86], a significant body of work has been done in both directions. Examples on the software prefetching side include prefetching for recursive data structures [87] and thread-based prefetching using speculative precomputation [88], and on the hardware side include streaming memory controllers to improve throughput by prefetching streams and using hardware prefetchers that utilize idle memory channel cycles by issuing prefetches [89]. A recent work on hardware prefetching focused on using smart stream detection methods that can dynamically modify prefetch settings to adapt to the workload needs [90]. [91] gives a survey on several alternative approaches to hardware and software data prefetching strategies.

Evaluation of Compiler Optimizations and Data Prefetching. [92] and [93] study the effect of compiler optimizations such as automatic vectorization, automatic parallelization, and inter-procedural optimization. These studies have not analyzed the impact of prefetching or streaming stores. The prefetching behavior of an Itanium 2 based SMP system was analyzed in [94]. While we work with a very different single-chip many-core co-processor architecture, we also evaluate the impact of streaming store instructions on this system.
Recently, [95] performed a detailed analysis of various software and hardware prefetching methods. Their goal was to provide a deeper understanding of the prefetching concept. They provide a detailed examination of the prefetch behavior of sequential SPEC CPU2006 applications using simulations of various out-of-order processor configurations. Our work differs from theirs from the following aspects. First, we target highly parallel and vectorized applications that put much more pressure on the memory system. Second, we evaluate the Intel MIC architecture with very different characteristics than their target systems. Third, our compiler generates software prefetch instructions automatically while they inserted prefetch instructions manually using intrinsics.

**Non-temporal and Streaming Stores.** Non-temporal and streaming store instructions appeared as a part of SIMD extensions. While SIMD extensions have been in mainframes in early 80s, they moved to desktop processors in late 90s with the goal of improving the performance of data-parallel applications. Popular examples of these extensions are SSE [96] from Intel, AltiVec [97] from Motorola, and 3DNow! [98] from AMD. An early work that evaluates the performance of various applications using SSE extensions is [99]. They use extensions such as prefetch and cache control instructions to improve performance. While the first extensions such as MMX, SSE, AltiVec used 128 bits, the latest industrial trend in SIMD extensions is to increase SIMD vector length. Recent processors with AVX use 256 bits vector operations and the Intel MIC architecture vector length is 512 bits. Hence, the importance of effective use of these SIMD extensions is increasing.

**Xeon Phi.** One of the early papers that presented results with Intel Xeon Phi was [78] from Intel. This work was using an Intel Xeon Phi Software Development Vehicle (formerly known as “Knights Ferry”) that was not publicly available, and it focused on generic parallelization, vectorization, and optimization features.
without any specific emphasis on Intel Xeon Phi. It did not include any detailed analysis on prefetching or streaming stores and the associated impact on this new architecture. A scalability study of graph algorithms on the same system was performed in [100]. However, the studies so far have approached from the angle of parallelization and vectorization of a single application and do not disclose any details on the overall prefetching and streaming store behavior. We believe that this is the first effort that analyzes the impact of compiler-based prefetching and streaming store instructions on a wide range of applications on Intel Xeon Phi product family. With the release of Intel Xeon Phi as a product, more studies that demonstrate the results of porting and optimizing applications to this new coprocessor will appear. These studies can use the software prefetch and streaming store generation mechanisms of the compiler presented in this work to improve the performance of their applications running on the Intel Xeon Phi coprocessor.

3.7 Conclusion

We provided details on compiler-based generation of software prefetch and streaming store instructions for the Intel Xeon Phi coprocessor. Our experiments showed that by exploiting these mechanisms to better hide memory latencies and save bandwidth, Intel Composer XE 2013 can achieve significant performance improvements. We believe that our findings will be useful for users of the Intel Xeon Phi coprocessor as well as users and compiler developers of other many-core vector architectures.
Summary and Future Work

4.1 Summary of Dissertation Contributions

This dissertation makes two major contributions towards addressing the memory problem in high performance computing systems.

The first contribution relates to the scalability problem of application-specific hardware accelerators. It is becoming more and more difficult to maintain coherence across memories in large-scale application-specific hardware accelerators, and the existing memory design approaches cannot scale to realize extreme-scale computing goals. To solve this scalability problem, a compiler-based automatic memory partitioning method is proposed in Chapter 2. This proposal generates an application-specific multi-level memory hierarchy for a given target application such that there is no need to use coherence hardware between sibling memories. As a result, memory systems that can scale seamlessly are designed, enabling the automatic synthesis of application-specific hardware accelerators for exascale computing.

The second contribution targets optimization of memory performance in gen-
eral purpose systems built using many-core processors. Due to the discrepancy between the improvements in processor and memory performance, modern many-core systems have very high memory access latencies and bandwidth contention. Due to these memory latency and bandwidth issues, integrating more cores onto the same die results in diminishing increases in performance. Effective methods to hide memory access latencies and reduce bandwidth demand of applications are required to continue enjoying performance improvements due to increased number of cores. To address these issues, Chapter 3 proposes and evaluates compiler-based methods to generate software prefetching and streaming vector store instructions. While software prefetching instructions hide memory access latencies by proactively pulling data from off-chip memory to the on-chip caches, streaming vector store instructions save precious off-chip memory bandwidth by eliminating redundant memory read operations.

4.2 Future Research Directions

There are abundant opportunities to extend and improve the ideas presented in this dissertation.

First, it is a fruitful direction to work towards the integration of the proposed automatic memory partitioning method with an application-specific accelerator synthesis framework and to test the quality of the generated memory hierarchies on actual hardware. This can be a very strong contribution to the field of automatic synthesis of scalable application-specific accelerators, and lead to various new, interesting research directions. Studying the interactions between the synthesized compute units and memory hierarchies, various improvements to the proposed memory partitioning method (e.g., generation of multi-banked, multi-ported mem-
ory hierarchies based on application requirements) can be performed. Further, the evaluation of the memory partitioning method in this work is restricted to a theoretical limit study, and therefore, no physical design constraints are imposed to the presented memory partitioner, in its current form. Investigating the impact of adding physical constraints such as die area, and analyzing the area-performance trade-offs in generating new child memories is also a very promising research direction.

Considering the optimization of memory performance in many-core processors, there is still a lot of room for research improving the coverage and accuracy of compiler-based software prefetching. For instance, automatically generating prefetch instructions for indirect memory references is still a very challenging task. Further, considering the significant bandwidth (and latency) benefits of using streaming store instructions, exploring alternative uses of these special instructions and implementing compiler optimizations that can automatically identify these cases can also be a very promising research area.
Bibliography


temporal Store Generation for Intel Xeon Phi Coprocessor," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2013 IEEE 27th International*.


tectural support for programming languages and operating systems, ASPLOS IV.


of OpenMP Programs,” in High Performance Computing - HiPC 2004, Lecture 
Notes in Computer Science, Springer Berlin Heidelberg, pp. 133–143.

compiler optimizations for the Intel Core 2 duo processor,” Int. J. Parallel 
Program., 36(6), pp. 571–591.

“Impact of Compiler-based Data-Prefetching Techniques on SPEC OMP Ap-
plication Performance,” in Proceedings of the 19th IEEE International Par-
allel and Distributed Processing Symposium, IPDPS ’05.


Streaming SIMD Extensions on the Pentium III Processor,” IEEE Micro, 

croprocessor Architecture,” Hot Chips 10.


Vita
Emre Kültürsay

Emre Kültürsay is PhD candidate in Computer Science and Engineering at the Pennsylvania State University. Emre holds a dual bachelor’s degree in Electrical and Electronics Engineering (EEE) and Computer Engineering (CEng) from the Middle East Technical University (METU, Turkey), and was a research assistant at the METU VLSI-MEMS laboratory between the years 2006 and 2008. After joining Penn State in 2008, he started serving an assistant to Prof. Mahmut Kandemir, conducting research on compilers and computer architecture. His primary research interests are compiler optimizations to improve memory performance of parallel architectures and the design of energy efficient, high performance computing systems. During his PhD, Emre also worked with a startup company under the supervision of Dr. Kemal Ebcioğlu for four years as a compiler developer, and with the Intel Corporation Compilers and Languages group under the supervision of Dr. Rakesh Krishnaiyer for nine months as a graduate intern. Emre’s research resulted in 14 publications in international conferences and journals, and two patent applications.