EXPLOITING SOFTWARE INFORMATION FOR AN EFFICIENT MEMORY HIERARCHY

BY

RAKESH KOMURAVELLI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Sarita V. Adve, Director of Research
Professor Marc Snir, Chair
Professor Vikram S. Adve
Professor Wen-mei W. Hwu
Dr. Ravi Iyer, Intel Labs
Dr. Gilles Pokam, Intel Labs
Dr. Pablo Montesinos, Qualcomm Research

ABSTRACT

Power consumption is one of the most important factors in the design of today's processor chips. Multicore and heterogeneous systems have emerged to address the rising power concerns. Since the memory hierarchy is becoming one of the major consumers of the on-chip power budget in these systems [73], designing an efficient memory hierarchy is critical to future systems. We identify three sources of inefficiencies in memory hierarchies of today's systems: (a) coherence, (b) data communication, and (c) data storage. This thesis takes the stand that many of these inefficiencies are a result of today's software-agnostic hardware design. There is a lot of information in the software that can be exploited to build an efficient memory hierarchy. This thesis focuses on identifying some of the inefficiencies related to each of the above three sources, and proposing various techniques to mitigate them by exploiting information from the software.

First, we focus on inefficiencies related to coherence and communication. Today's hardware based directory coherence protocols are extremely complex and incur unnecessary overheads for sending invalidation messages and maintaining sharer lists. We propose DeNovo, a hardware-software co-designed protocol, to address these issues for a class of programs that are deterministic. DeNovo assumes a disciplined programming environment and exploits features such as structured parallel control, data-race-freedom, and software information about data access patterns to build a system that is simple, extensible, and performance-efficient compared to today's protocols. We also extend DeNovo to add two optimizations to address the inefficiencies related to data communication, specifically, aimed at reducing the unnecessary on-chip network traffic. We show that adding these two optimizations did not only result in addition of zero new states (or transient states) to the protocol but also provided performance and energy gains to the system, thus validating the extensibility of the DeNovo protocol. Together with the two communication optimizations DeNovo reduces the memory stall time by 32% and the network traffic by 36% (resulting in direct savings in energy) on average compared to a state-of-the-art implementation of the MESI protocol for the applications studied.

Next we address the inefficiencies related to data storage. Caches and scratchpads are two popular organizations for storing data in today's systems but they both have inefficiencies. Caches are power-hungry incurring expensive tag lookups and scratchpads incur unnecessary data movement as they are only locally visible. To address these problems, we propose a new memory organization, stash, which has the best of both cache and scratchpad organizations. Stash is a globally visible unit and its functionality is independent of the coherence protocol employed. In our implementation, we extend DeNovo to provide coherence for stash. Compared to a baseline configuration that has both scratchpad and cache accesses, we show that the stash configuration (in which scratchpad and cache accesses are converted to stash accesses), even with today's applications that do not fully exploit stash, reduces the execution time by 10% and the energy consumption by 14% on average.

Overall, this thesis shows that a software-aware hardware design can effectively address many of the inefficiencies found in today's software oblivious memory hierarchies.

To my parents and my brother

ACKNOWLEDGMENTS

My Ph.D. journey has been long. There are several people whom I would like to thank for supporting me and believing in me throughout the journey.

First and the foremost, I want to thank my advisor, Sarita Adve, for giving me the opportunity to pursue my dream of getting a Ph.D. I am very much grateful for her constant guidance and support for the past six years and for making me a better researcher. Her immense enthusiasm for research, her never-give-up attitude, and her always striving for the best quality are some of the traits that will inspire and motivate me forever. I am truly honored to have Sarita as my Ph.D. advisor.

I would also like to thank Vikram Adve for his constant guidance on the DeNovo and the stash projects. If there were any formal designation, he would perfectly fit the description of a co-advisor. I thank Nick Carter and Ching-Tsun Chou for their collaborations on the DeNovo project and Pablo Montesinos for his collaboration on the stash project. I also sincerely thank the rest of my Ph.D. committee, Marc Snir, Wen-Mei Hwu, Ravi Iyer, and Gilles Pokam for their insightful comments and suggestions for improvements on my thesis. Special thanks to Bhushan Chitlur, my internship mentor at Intel, for exposing me to the real world architecture problems and experience.

I am thankful to Hyojin Sung and Byn Choi for the collaborations on the DeNovo project and to Matt Sinclair for the collaborations on the stash project. I have learned a lot by working closely with these three folks. In addition, I am also thankful for my other collaborators on these projects, Rob Bocchino, Nima Honarmand, Rob Smolinski, Prakalp Srivastava, Maria Kotsifakou, John Alsop, and Huzaifa Muhammad. I thank my other lab-mates, Pradeep Ramachandran, Siva Hari, Radha Venkatagiri, and Abdulrahman Mahmoud who played a very important role in not only providing a great and fun research environment but also aiding me in intellectual development.

I thank the Computer Science department at Illinois for providing a wonderful Ph.D. curriculum and flexibility for conducting research. Specifically, I would like to thank the staff members, Molly, Andrea,

Michelle, and Mary Beth who on several occasions helped me with administrative chores.

My work has been supported by Intel and Microsoft through the Universal Parallel Computing Research Center (UPCRC) at Illinois, by Intel through the Illinois/Intel Parallelism Center at Illinois, by the National Science Foundation under grants CCF-1018796 and CCF-1302641, and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. My work was also supported by a Qualcomm Innovation Fellowship.

My immense thanks to the numerous friends and acquaintances I have made here in the cities of Urbana and Champaign with whom I have experienced some of my life's best moments. I was surrounded by people from all walks of life and cultures from all over the world. This helped me grow in life outside of work and become a much better person than I was before coming to the United States. I will be forever thankful to Joe Grohens and the tango dance community here for introducing me to the most amazing dance and Meg Tyler for teaching me how to ride horses and helping me pursue my childhood dream. I also thank Porscha, Tajal, Steph, Adi, Tush, Susan, Ankit, Natalie, and Amit for their friendships and for all the fun times.

Finally, it goes without saying how much I am indebted to my parents and my brother for their understanding and encouragement for all these years. This journey of my Ph.D. would have been much harder if it were not for you and I proudly dedicate this dissertation to you.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation

Recent advances in semiconductor technology have helped Moore's law to continue. In the past, when leakage current was minimal, increased chip densities accompanied with supply voltage scaling resulted in constant power consumption for a given area of the chip. Unfortunately, with the recent breakdown of the classical CMOS voltage scaling, power has become a first class problem in the design of processor chips leading to new research directions in the field of computer architecture.

Multicores are one such attempt to address the rising power consumption problem. Alternately, heterogeneous systems take a different approach where power efficient individual components (e.g., GPU, DSP, FPGA, accelerators, etc.) are specialized for various problem domains as opposed to a general-purpose homogeneous multicore system. However, these specialized components differ in many aspects including ISAs, functionality, and underlying memory models and hierarchy. These differences imply difficulty in building a power efficient heterogeneous system that can be effectively used. Both standalone multicores and a cluster of specialized components have their own advantages and disadvantages. Hence we are increasingly seeing the trend towards hybrid systems which have part multicore and part specialized components [82, 73, 32, 68].

With the rise of such hybrid systems, today's computer systems, from smartphones to servers, are more complex than ever before. Data movement in these systems is expected to become the dominant consumer of energy as technology continues to scale [73]. For example, a recent study has shown that by 2017 more than 50% of the total energy for a 64-bit GPU floating-point computation will be spent in the memory access (reading three source operands and writing to a destination operand from/to an 8KB SRAM) [73].

This highlights the urgent need for minimizing data movement and an energy-efficient memory hierarchy for future scalable computer systems.

Shared-memory is arguably the most widely used parallel programming model. Today's shared-memory hierarchies have several inefficiencies. In this thesis, we focus on homogeneous multicores and heterogeneous SoC systems. In multicores, complex directory-based coherence protocols, inefficient data transfers, and power-inefficient caches make it hard to design performance-, power-, and complexity-scalable hardware. These inefficiencies are exacerbated as more and more cores are added to the system. Traditionally, memory units of different components in heterogeneous SoC systems are only *loosely coupled* with respect to one another. Any communication between the components required interaction through main memory, which incurs unnecessary data movement and latency overheads. Recent designs such as AMD's Fusion [32] and Intel's Haswell [68] address this issue by creating more *tightly coupled* systems with a single unified address space and coherent caches. By tightly coupling the cores, data can be sent from one component to another without needing the explicit transfer through the main memory. However, these architectures have other inefficiencies in the memory hierarchy. For example, these systems provide only partial coherence and local memories are not globally accessible.

Many of these problems of shared-memory systems are because of today's software agnostic hardware design. They can be mitigated by having more disciplined programming models and by exploiting the information that is already available in the software. Many of today's undisciplined programming models allow arbitrary reads and writes for implicit and unstructured communication and synchronization. This results in "wild shared-memory" behaviors with unintended data races and non-determinism and implicit side effects. The same phenomena result in complex hardware that must assume that any memory access may trigger communication, and performance- and power-inefficient hardware that is unable to exploit communication patterns known to the programmer but obfuscated by the programming model. There is much recent software work on more *disciplined* shared-memory programming models to address the above problems. We believe that exploiting the guarantees provided by such disciplined programming models will help us alleviate some of the inefficiencies in the memory hierarchy. Also applications have a lot of other information that could be utilized by the hardware to be more efficient. Applications for heterogeneous systems (e.g., using CUDA and OpenCL programming models) have additional information like which data is communicated between the CPU and the accelerator, which parts of the main memory are explicitly assigned to a local scratchpad, which data is read only, and so on. Such information (if available to the hardware) can be exploited to design efficient data communication and storage. Hence software-aware hardware that exploits information from the software will help us rethink today's memory hierarchy to achieve energy-efficient and complexity-scalable hardware.

1.2 Inefficiencies in Today's Memory Hierarchies

We identify three broad classes of problems with today's shared-memory systems:

Inefficiencies with techniques used for sharing data (a.k.a. coherence protocols): Hardware based directory coherence protocols used in today's shared memory systems have several limitations. They are extremely complex and incur high verification overhead because of numerous transient states and subtle races; incur additional traffic for invalidation and acknowledgement messages; incur high storage overhead to keep track of sharer lists; and suffer from false sharing due to aggregated cache state.

Inefficiencies with how data is communicated: Today's cache-line granularity data transfers are not always optimal. Cache line transfers are easy to implement but incur additional network traffic for unused words in the cache line. Moreover, traditional request and response traffic that flows through the cache hierarchy and mandatory hop at the directory may not be always required (e.g. read-only streaming data, known producer, and so on).

Inefficiencies with how data is stored: Caches and scratchpads are two common types of memory organizations that today's memory hierarchies support. Caches are easy to program (largely invisible to the programmer) but are power inefficient due to tag lookups and misses. They also store data at cache line granularity which is not always optimal. Scratchpads, in contrast, are energy- and delay-efficient compared to caches with their guaranteed hits. But scratchpads are only locally visible (requiring explicit programmer support) and hence need explicit copying of data from and to main memory. This typically results in explicit data movement, executing additional instructions, usage of core's registers, incurring additional network traffic, and polluting the cache.

1.3 Contributions of this Thesis

In this thesis, we analyze each of the above three types of memory hierarchy inefficiencies, find ways to exploit information available in software, and propose solutions to mitigate them to make hardware more energy-efficient. We limit our focus to deterministic codes in this thesis for multiple reasons: (1) There is a growing view that deterministic algorithms will be common, at least for client-side computing [1]; (2) focusing on these codes allows us to investigate the "best case;" i.e., the potential gain from exploiting strong discipline; (3) these investigations form a basis to develop the extensions needed for other classes of codes (pursued partly for this thesis and partly by other members of the larger project). Synchronization mechanisms involve races and are used in all classes of codes; in this thesis, we assume special techniques to implement them (e.g., hardware barriers, queue based locks, etc.). Their detailed handling is explored by the larger project (some of this work is described below) and is not part of this thesis. The specific contributions of this thesis are as follows.

1.3.1 DeNovo: Addressing Coherence and Communication Inefficiencies

DeNovo [45] addresses the many inefficiencies of today's hardware based directory coherence protocols.[1] It assumes a disciplined programming environment and exploits properties of such environments like structured parallel control, data-race-freedom, deterministic execution, and software information about which data is shared and when. DeNovo uses Deterministic Parallel Java (DPJ) [28, 29] as an exemplar disciplined language providing these properties. Two key insights underlie DeNovo's design. First, structured parallel control and knowing which memory regions will be read or written enable a cache to take responsibility for invalidating its own stale data. Such self-invalidations remove the need for a hardware directory to track sharer lists and to send invalidations and acknowledgements on writes. Second, data-race-freedom eliminates concurrent conflicting accesses and corresponding transient states in coherence protocols, eliminating a major source of complexity. Specifically, DeNovo provides the following benefits.

Simplicity: To provide quantitative evidence of the simplicity of the DeNovo protocol, we compared it with a conventional MESI protocol [108] by implementing both in the Murphi model checking tool [54]. For MESI, we used the implementation in the Wisconsin GEMS simulation suite [94] as an example of a

[1] I co-led the design and evaluation of the DeNovo protocol with my colleagues, Byn Choi and Hyojin Sung [45]. This work will also appear in Hyojin Sung's thesis. I was solely responsible for the verification work for the DeNovo protocol [78]. This work also appears in my M.S. thesis and is presented here for completeness.

(publicly available) state-of-the-art, mature implementation. We found several bugs in MESI that involved subtle data races and took several days to debug and fix. The debugged MESI showed 15X more reachable states compared to DeNovo, with a verification time difference of 173 seconds vs 8.66 seconds [78]. These results attest to the complexity of the MESI protocol and the relative simplicity of DeNovo.

Extensibility: To demonstrate the extensibility of the DeNovo protocol, we implemented two optimizations addressing inefficiencies related to data communication: (1) Direct cache-to-cache transfer: Data in a remote cache may directly be sent to another cache without indirection to the shared lower level cache (or directory). (2) Flexible communication granularity: Instead of always sending a fixed cache line in response to a demand read, we send a programmer directed set of data associated with the region information of the demand read. Neither optimization required adding any new protocol states to DeNovo; since there are no sharer lists, valid data can be freely transferred from one cache to another.

Storage overhead: The DeNovo protocol incurs no storage overhead for directory information. But we need to maintain coherence state bits and additional information at the granularity at which we guarantee data-race freedom, which can be less than a cache line. For low core counts, this overhead is higher than with conventional directory schemes, but it pays off after a few tens of cores and is scalable (constant per cache line). A positive side effect is that it is easy to eliminate the requirement of inclusivity in a shared last level cache (since we no longer track sharer lists). Thus, DeNovo allows more effective use of shared cache space.

Performance and power: In our evaluations, we show that the DeNovo coherence protocol along with the communication optimizations described above reduces an average 32% (up to 77%) of the memory stall time and an average reduction of 36% (up to 71.5%) of the network traffic compared to MESI. The reductions in network traffic have direct implications on energy savings.

1.3.2 Stash: Addressing Storage Inefficiencies

The memory hierarchies of heterogeneous SoCs are often loosely coupled and require explicit communication through main memory to interact. This results in unnecessary data movement and latency overheads. A more tightly coupled SoC memory hierarchy helps address these problems, but doesn't remove all sources of inefficiency such as power-inefficient cache accesses and scratchpads that are only locally visible. To combat this, we introduce a new memory organization called a **stash**[2] [79] that has the best properties of both scratchpads and caches. Similar to a scratchpad, stash is software managed, directly addressable, and provides compact data storage. Stash also has a mapping between the global and stash address spaces. This helps stash to be globally visible and replicate data like a cache. Replication needs support for coherence and any existing protocol can be extended to support stash. In this thesis, we extend the simple and efficient DeNovo protocol to support coherence for stash. Our results show that, compared to a baseline configuration that has both scratchpad and global cache accesses, the stash configuration (that converts all scratchpad and global accesses to stash accesses) reduces the execution time by 10% and the energy consumption by 14% on average.

1.4 Other Contributions

I have contributed to some other works in the larger project that this thesis is a part of but are not included in this thesis. This section provides a brief summary of these works.

1.4.1 Understanding the Properties of Disciplined Software

The DeNovo protocol introduced above exploits several properties of a disciplined programming environment. To understand these properties well and explore how to exploit them in hardware, we studied the language and also actively contributed to the evaluations of DPJ [28, 29], the driver language for DeNovo. Specifically, I have ported several applications to DPJ and performed application analysis to understand what information could be exploited in hardware.

1.4.2 DeNovoND: Support for Disciplined Non-determinism

DeNovo focuses on a class of programs that are deterministic. DeNovoND [130, 131] takes a step forward and extends DeNovo to support programs with disciplined non-determinism. DPJ permits disciplined non-determinism by permitting conflicting accesses, but constraining them to occur within well defined atomic sections with explicitly declared atomic regions and effects [29]. We have shown that modest extensions to DeNovo can allow this form of non-determinism without sacrificing its advantages. The resulting system, DeNovoND, provides comparable or better performance than MESI for several applications designed for

[2] I co-led the work on stash with my colleague, Matthew D. Sinclair.

lock synchronization, and shows 33% less network traffic on average, implying potential energy savings. My specific contributions to DeNovoND are designing and implementing queue based locks in hardware.

1.5 Outline of the Thesis

This thesis is organized as follows. Chapter 2 describes our solutions to address the coherence and communication inefficiencies. In this chapter, we describe the DeNovo coherence protocol and the two communication optimizations that extend DeNovo. Chapter 3 provides a complexity analysis of DeNovo by formally verifying it and comparing the effort against that of a state-of-the-art implementation of MESI. We provide performance analysis of DeNovo in Chapter 4. In Chapter 5, we introduce stash that addresses the storage inefficiencies. We provide performance evaluation of the stash organization in Chapter 6. Chapter 7 describes the prior work. Finally, Chapter 8 summarizes the thesis and provides directions for future work.

1.6 Summary

On-chip energy has become one of the primary constraints in building computer systems. Today's complex and software-oblivious systems have several inefficiencies which are hindrances for building future energy-efficient systems. This thesis takes the stand that there is a lot of information in the software that can be exploited to remove these inefficiencies. We focus on three sources of inefficiencies in today's memory hierarchies: (a) coherence, (b) data communication, and (c) data storage.

Specifically, we propose a simple and scalable hardware-software co-designed DeNovo coherence protocol to address inefficiencies in today's complex hardware directory based protocols. We extend DeNovo with two optimizations that are aimed at reducing the unnecessary on-chip network traffic addressing the inefficiencies in data communication. Finally, to address several inefficiencies with data storage, we propose a new memory organization, $stash$, that has the best of both scratchpad and cache organizations.

Together, we show that a true software-hardware co-designed system that exploits information from software makes for an efficient system compared to today's largely software-oblivious systems.

CHAPTER 2

COHERENCE AND COMMUNICATION

In a shared-memory system, coherence is required when multiple compute units (homogeneous or heterogeneous) replicate and modify the same data. Coherence is usually associated with cache memory organization. But similar to caches, there are other memory organizations like *stash*, as described in Chapter 5, that hold globally addressable and replicable data, which require coherence too. Shared-memory systems typically implement coherence with snooping or directory-based protocols in the hardware. Although current directory-based protocols are more scalable than snooping protocols, they suffer from several limitations:

Performance and power overhead: They incur several sources of latency and traffic overhead, impacting performance and power; e.g., they require invalidation and acknowledgment messages (which are strictly overhead) and indirection through the directory for cache-to-cache transfers.

Verification complexity and extensibility: They are notoriously complex and difficult to verify since they require dealing with subtle races and many transient states (Section 2.1.2) [103, 60]. Furthermore, their fragility often discourages implementors from adding optimizations to previously verified protocols – additions usually require re-verification due to even more states and races.

State overhead: Directory protocols incur high directory storage overhead to track sharer lists. Several optimized directory organizations have been proposed, but also require considerable overhead and/or excessive network traffic and/or complexity. These protocols also require several coherence state bits due to the large number of protocol states (e.g., ten bits in [115]). This state overhead is amortized by tracking coherence at the granularity of cache lines. This can result in performance/power anomalies and inefficiencies when the granularity of sharing is different from a contiguous cache line (e.g., false sharing).

Researchers continue to propose new hardware directory organizations and protocol optimizations to address one or more of the above limitations (Section 7.1); however, all of these approaches incur one or more of complexity, performance, power, or storage overhead. In this chapter, we describe DeNovo, a hardware-software co-designed approach, that exploits emerging disciplined software properties in addition to data-race-freedom to target all the above mentioned limitations of directory protocols for large core counts. Next, we describe these disciplined software properties that DeNovo exploits and some insight into how complex today's hardware protocols are.

2.1 Background

2.1.1 Disciplined Parallel Models and Deterministic Parallel Java (DPJ)

There has been much recent research on disciplined shared-memory programming models with explicit and structured communication and synchronization for both deterministic and non-deterministic algorithms [1]; e.g., Ct [59], CnC [33], Cilk++ [26], Galois [81], SharC [11], Kendo [107], Prometheus [9], Grace [21], Axum [61], and Deterministic Parallel Java (DPJ) [28, 29].

We employ Deterministic Parallel Java (DPJ) [28] as an exemplar of the emerging class of deterministic-by-default languages, and use it to explore how hardware can take advantage of strong disciplined programming features. Specifically, we use three features of DPJ that are also common to several other projects: (1) structured parallel control; (2) data-race-freedom, and guaranteed deterministic semantics unless the programmer explicitly requests non-determinism (called determinism-by-default); and (3) explicit specification of the side effects of parallel sections; e.g., which (possibly non-contiguous) regions of shared-memory will be read or written in a parallel section.

Most of the disciplined models projects cited above also enforce a requirement of structured parallel control (e.g., a nested fork join model, pipelining, etc.), which is much easier to reason about than arbitrary (unstructured) thread synchronization. Most of these guarantee the absence of data races for programs that type-check. Coupled with structured parallel control, the data-race-freedom property guarantees determinism for several of these systems. We also note that data races are prohibited (although not checked) by existing popular languages as well; the latest C++ and C memory models [30] do not provide *any* semantics with any data race (benign or otherwise) and Java [92] provides extremely complex and weak semantics for data races only for the purposes of ensuring safety. The information about side effects of concurrent tasks is also available in other disciplined languages, but in widely varying (and sometimes indirect) ways.

DPJ is an extension to Java that enforces *deterministic-by-default* semantics via compile-time type checking [28, 29]. Using Java is not essential; similar extensions for C++ are possible. DPJ provides a new type and effect system for expressing important patterns of deterministic and non-deterministic parallelism in imperative, object-oriented programs. Non-deterministic behavior can only be obtained via certain explicit constructs. For a program that does not use such constructs, DPJ guarantees that if the program is well-typed, any two parallel tasks are *non-interfering*, i.e., do not have conflicting accesses. (Two accesses conflict if they reference the same location and at least one is a write.)

DPJ's parallel tasks are iterations of an explicitly parallel `foreach` loop or statements within a `cobegin` block; they synchronize through an implicit barrier at the end of the loop or block. Parallel control flow thus follows a scoped, nested, fork-join structure, which simplifies the use of explicit coherence actions in DeNovo at fork/join points. This structure defines a natural ordering of the tasks, as well as an obvious definition of when two tasks are "concurrent". It implies an obvious sequential equivalent of the parallel program (`for` replaces `foreach` and `cobegin` is simply ignored). DPJ guarantees that the result of a parallel execution is the same as the sequential equivalent.

In a DPJ program, the programmer assigns every object field or array element to a named "*region*" and annotates every method with read or write "*effects*" summarizing the regions read or written by that method. The compiler checks that (i) all program operations are type safe in the region type system; (ii) a method's effect summaries are a superset of the actual effects in the method body; and (iii) that no two parallel statements interfere. The effect summaries on method interfaces allow all these checks to be performed without interprocedural analysis.

For DeNovo, the effect information tells the hardware what fields will be read or written in each parallel "phase" (`foreach` or `cobegin`). This enables efficient software-controlled coherence mechanisms discussed in the following sections.

DPJ has been evaluated on a wide range of deterministic parallel programs. The results show that DPJ can express a wide range of realistic parallel algorithms; that its type system features are useful for such programs; and that well-tuned DPJ programs exhibit good performance [28].

In addition to guaranteeing determinism, DPJ was later extended to provide strong safety properties such as data-race-freedom, strong isolation, and composition for non-deterministic code sections [29]. This is achieved by ensuring that conflicting accesses in concurrent tasks are confined to *atomic sections* and their regions and effects are explicitly annotated as *atomic*. In this thesis, we focus only on the deterministic codes.

2.1.2 Complexity of Traditional Coherence Protocols

To understand the complexity of today's directory-based protocols [78], we briefly discuss the details of a state-of-the-art, mature, publicly available protocol, the MESI protocol implemented in the Wisconsin GEMS simulation suite (version 2.1.1) [94]. Without loss of generality, we assume a multicore system with n cores, private $L1$ caches, a shared $L2$ cache, and a general (non-bus, unordered) interconnect on chip.

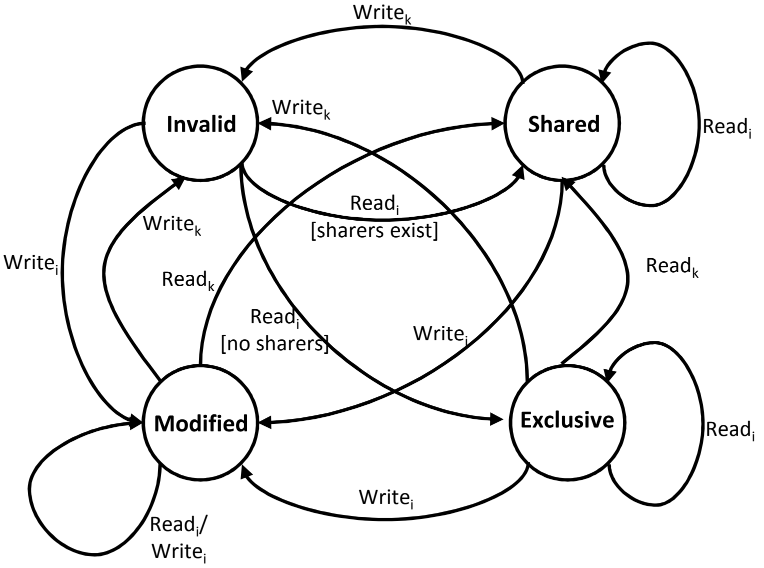


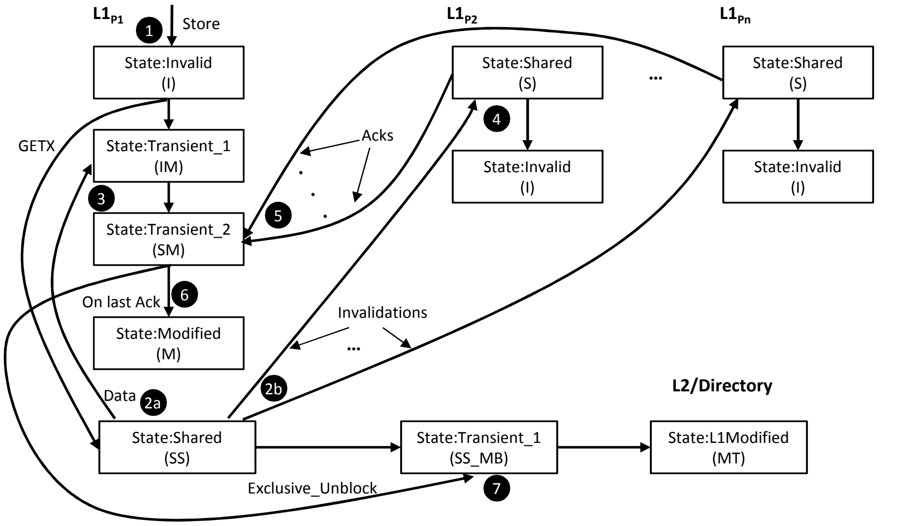
Figure 2.1 Textbook state transition diagram for $L1$ cache of core i for the MESI protocol. $Read_i$ = read from core i, $Read_k$ = read from another core k.

MESI, also known as the Illinois protocol [108], stands for $Modified$ (locally modified and no other cache has a copy), $Exclusive$ (unmodified and no other cache has a copy), $Shared$ (unmodified and some other caches may have a copy), and $Invalid.$ Over the MSI protocol, the $Exclusive$ state has the added advantage of avoiding invalidation traffic on write hits. For scalability, we assume a directory protocol [86]. Given our shared (inclusive) $L2$ cache based multicore, we assume a directory entry per $L2$ cache line, referred to as an in-cache directory [39]. We use $L2$ and $directory$ interchangeably.

Figure 2.1 shows the simple textbook state transition diagram for an $L1$ cache with the MESI protocol. The $L2$ cache also has four (textbook) states, $L1\_Modified$ (modified in a local $L1$), $L2\_Modified$

```
a_type A;
//phase 1
parallel_for{
   …
   //P2 … Pn
   if(not coreP1) {
      Read A;
   }
   …
}
//implicit barrier
//phase 2
parallel _for{
   …
   //only P1
   if(coreP1) {
      Write A;     ←
   }
   …
}
//implicit barrier
```

(a) Example parallel code



(b) Timeline of state transitions at L1 and L2

State/Event	Write_i	Ack	Ack_all	Data	Data_Ack_all

Invalid
(I)
	Go to Transient_1;
	GETX to L2;
		X	X	X	X

Transient_1
(IM)
	Keep pending;	Update Ack count;	X	Update Ack count;
				Go to Transient_2;
					Go to Modified;
					Exclusive_Unblock to L2;

Transient_2
(SM)
	Keep pending;	Update Ack count;	Go to Modified;
			Exclusive_Unblock to L2;
				X	X

(c) L1 state transition table

State/Event	L1_GETX	Exclusive_Unblock

Shared
(SS)
	Go to Transient_1;
	Ack to requestor;
	Inv to sharers;
		X

Transient_1 (SS_MB)	--	Go to Modified (MT)

(d) L2 state transition table

Figure 2.2 Example state transitions for MESI.

(modified at $L2$ and not present in any $L1$), $Shared$ (valid data at $L2$ and present in one or more $L1$s) and $Invalid$. It also has a $dirty$ bit (set on receiving a writeback from $L1$) which indicates whether data is dirty or not. When in $Shared$ state, the $L2/directory$ contains the up-to-date copy of the cache block data along with a list of sharers. On a read miss request, the $directory$ services the request if it has the up-to-date copy, or else it forwards the request to the core that has the exclusive or modified copy. On a write miss or upgrade request, the $directory$ sends invalidation requests to all the sharers (if any). If a request misses in the $L2$, the block is fetched from the main memory.

In reality, this seemingly simple protocol is a lot more complex. The hardware implementation of the protocol has many transient states in addition to the four states described above. These transient states lead to various subtle races and are the root cause of the complexity in the protocol. We now illustrate the need for transient states with an example. Figure 2.2(a) shows a code snippet with two parallel phases accessing a shared variable A.

In the first phase, cores $P2$ through Pn read the shared variable A and in the second phase, core $P1$ writes to A. In this example, we focus mainly on the state transitions related to this write by core $P1$.

Figure 2.2(b) shows the timeline of the state transitions at both the individual $L1$s and the $L2$. Figures 2.2(c) and 2.2(d) show the state transition table for $L1$ and $L2$ respectively for the states encountered in this example. The names of the states and the events are taken directly from the GEMS implementation. At the beginning of the second phase, cores $P2$ through Pn are in $Shared$ state and they are recorded in the sharer list at the $directory$. On receiving the write request, $L1_{P1}$ issues a $GETX$ request to $L2$ and transitions to the first transient state, IM, where it awaits the data response from $L2$. $L2$, on receiving the $GETX$ request, sends the data response (including the number of Acks to expect) to $L1_{P1}$, sends invalidation requests to all the sharers ($L1_{P2}$ through $L1_{Pn}$), and then transitions to a transient state, SS_MB, where it awaits an unblock message from the original requestor indicating the completion of the request. $L1_{P1}$, on receiving the data response from $L2$, transitions to the second transient state, SM, where it waits for all the Ack messages from the sharers. Every sharer, then, on receiving the invalidation message from $L2$, transitions to the $Invalid$ state and responds directly to the requestor, $L1_{P1}$, with an Ack message. When $L1_{P1}$ receives the last Ack message (Ack_all event), it transitions to the $Modified$ state and unblocks $L2$ by sending an $Exclusive\_Unblock$ message. Other cases (e.g., some of the Acks arriving at $L1_{P1}$ before it receives the data response from $L2$, etc.) are covered in the state transition tables shown in Figures 2.2(c) and 2.2(d). Ack_all is triggered for the last incoming Ack and $Data\_Ack\_all$ is triggered if $L2$'s data response (which includes the Ack count) is the last message to be received.

This example illustrates the need for transient states and the additional complexities introduced by them in the MESI protocol. The larger the number of transient states, the more complex the protocol becomes. In the GEMS implementation of the MESI protocol, there are seven transient states in $L1$ and 14 transient states in $L2$. Optimizations to the above baseline protocol usually incur additional transient states.

2.2 DeNovo

In this section, we describe how we exploit various features of disciplined programming languages mentioned earlier to redesign a hardware coherence protocol that is not only complexity-efficient but also energy- and performance-efficient compared to traditional protocols.

A shared-memory design must first and foremost ensure that a read returns the correct value, where the definition of “correct” comes from the memory consistency model. Modern systems divide this responsibility between two parts: (i) cache coherence, and (ii) various memory ordering constraints. These are arguably among the most complex and hard to scale aspects of shared-memory hierarchy design. Disciplined models enable mechanisms that are potentially simpler and more efficient to achieve this function.

The deterministic parts of our software have semantics corresponding to those of the equivalent sequential program. A read should therefore simply return the value of the last write to the same location that is before it in the deterministic sequential program order. This write either comes from the reader’s own task (if such a write exists) or from a task preceding the reader’s task, since there can be no conflicting accesses concurrent with the reader (two accesses are concurrent if they are from concurrent tasks). In contrast, conventional (software-oblivious) cache coherence protocols assume that writes and reads to the same location can happen concurrently, resulting in significant complexity and inefficiency.

To describe the DeNovo protocol, we first assume that the coherence granularity and address/communication granularity are the same. That is, the data size for which coherence state is maintained is the same as the data size corresponding to an address tag in the cache and the size communicated on a demand miss. This is typically the case for MESI protocols, where the cache line size (e.g., 64 bytes) serves as the address, communication, and coherence granularity. For DeNovo, the coherence granularity is dictated by the granularity at which data-race-freedom is ensured – a word for our applications. Thus, this assumption constrains the cache line size. We henceforth refer to this as the word based version of our protocol. We relax this assumption in Section 2.2.2, where we decouple the address/communication and coherence granularities and also enable sub-word coherence granularity.

Without loss of generality, throughout we assume private and writeback L1 caches, a shared last-level on-chip L2 cache inclusive of only the modified lines in any L1, a single (multicore) processor chip system, and no task migration. The ideas here extend in an obvious way to deeper hierarchies with multiple private and/or cluster caches and multichip multiprocessors, and task migration can be accommodated with appropriate self-invalidations before migration. Below, we use the term *phase* to refer to the execution of all tasks created by a single parallel construct (foreach or cobegin).

2.2.1 DeNovo with Equal Address/Communication and Coherence Granularity

DeNovo eliminates the drawbacks of conventional directory protocols as follows.

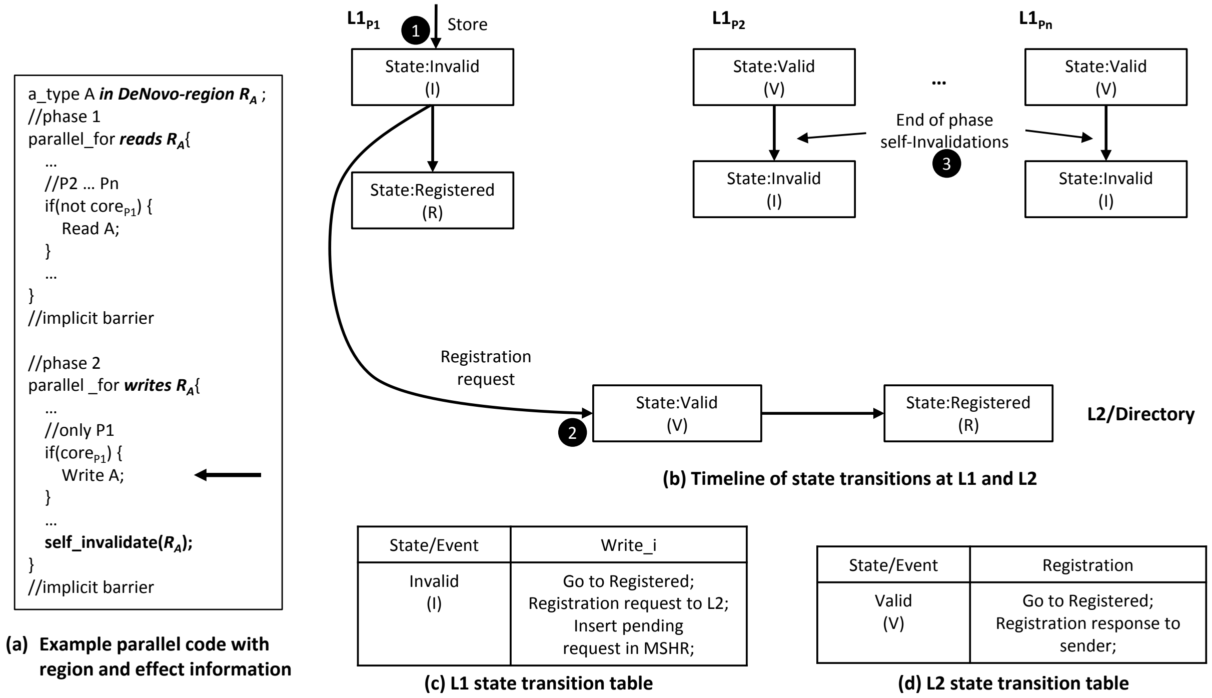
No directory storage or write invalidation overhead: In conventional directory protocols, a write acquires ownership of a line by invalidating all other copies, to ensure later reads get the updated value. The directory achieves this by tracking all current sharers and invalidating them on a write, incurring significant storage and invalidation traffic overhead. In particular, straightforward bit vector implementations of sharer lists are not scalable. Several techniques have been proposed to reduce this overhead, but typically pay a price in significant increase in complexity and/or incurring unnecessary invalidations when the directory overflows. DeNovo eliminates these overheads by removing the need for invalidations on a write. Data-race-freedom ensures there is no other writer or reader for that line in this parallel phase. DeNovo need to only ensure that (i) outdated cache copies are invalidated before the next phase, and (ii) readers in later phases know where to get the new data.

For (i), each cache simply uses the known write effects of the current phase to invalidate its outdated data before the next phase begins. The compiler inserts self-invalidation instructions for each region with these write effects (we describe how regions are conveyed and represented below). Each L1 cache invalidates its data that belongs to these regions with the following exception. Any data that the cache has read or written in this phase is known to be up-to-date since there cannot be concurrent writers. We therefore augment each line with a "touched" bit that is set on a read. A self-invalidation instruction does not invalidate a line with a set touched bit or that was last written by this core (indicated by the `registered` state as discussed below); the instruction resets the touched bit in preparation for the next phase.

For (ii), DeNovo requires that on a write, a core register itself at (i.e., inform) the shared L2 cache. The L2 data banks serve as the registry. An entry in the L2 data bank either keeps the identity of an L1 that has the up-to-date data (`registered` state) or the data itself (`valid` state) – a data bank entry is never required to keep both pieces of information since an L1 cache registers itself in precisely the case where the L2 data bank does not have the up-to-date data. Thus, DeNovo entails *zero overhead for directory (registry) storage*. Henceforth, we use the term L2 cache and registry interchangeably.

We also note that because the L2 does not need sharer lists, it is natural to not maintain inclusion in the L2 for lines that are not registered by another L1 cache – the registered lines do need space in the L2 to track the L1 id that registered them.

No transient states: The DeNovo protocol has three states in the L1 and L2 – `registered`, `valid`, and `invalid` – with obvious meaning. (The touched bit mentioned above is local to its cache and irrelevant to external coherence transactions.) As described in Section 2.1.2, conventional directory protocols require several transient states making them notoriously complex and difficult to verify [4, 125, 140]. DeNovo, in contrast, is a true 3-state protocol with *no transient states*, since it assumes race-free software. The only possible races are related to writebacks. As discussed below, these races either have limited scope or are similar to those that occur in uniprocessors. They can be handled in straightforward ways, without transient protocol states (described below).



```
a_type A in DeNovo-region R_A ;
//phase 1
parallel_for reads R_A{
   ...
   //P2 ... Pn
   if(not core_P1) {
      Read A;
   }
   ...
}
//implicit barrier

//phase 2
parallel _for writes R_A{
   ...
   //only P1
   if(core_P1) {
      Write A;
   }
   ...
   self_invalidate(R_A);
}
//implicit barrier
```

(a) Example parallel code with region and effect information

(b) Timeline of state transitions at L1 and L2

State/Event	Write_i

Invalid
(I)
	Go to Registered;
	Registration request to L2;
	Insert pending request in MSHR;

(c) L1 state transition table

State/Event	Registration
Valid (V)	Go to Registered; Registration response to sender;

(d) L2 state transition table

Figure 2.3 Example state transitions for DeNovo.

Let us revisit the code segment from Figure 2.2. Figure 2.3(a) shows the changes to the code required to prove data-race-freedom. Specifically, the shared variable A is placed in a region R_A, both the parallel phases are annotated with read and write effect summaries, and finally a self-invalidation instruction is inserted at the end of the second phase.

Figure 2.3(b) shows the timeline of the state transitions for the DeNovo protocol and the state transition tables for the states encountered in this example are shown in Figures 2.3(c) and 2.3(d). Focusing again on the write instruction in the second phase, $L1_{P1}$ transitions directly to the $Registered$ state without transitioning to any transient state and sends a registration request to $L2$. $L2$, on receiving the registration request, transitions to the $Registered$ state. We do not show the registration response message from $L2$

here as it is not in the critical path and is handled by the request buffer at $L1$. At the end of the phase, each core executes a self-invalidate instruction on region R_A. This instruction triggers the invalidation of all the data in region R_A in the $L1$ cache of its core except for data in $Registered$ state and that is both $Valid$ and $touched$, since this data is known to be up-to-date. The $touched$ bits are reset at the end of the parallel phase. This example illustrates how the absence of transient states makes the DeNovo protocol simpler than MESI.

The full protocol: Table 2.1 shows the L1 and L2 state transitions and events for the full protocol. Note the lack of transient states in the caches.

Read requests to the L1 (from L1's core) are straightforward – accesses to valid and registered state are hits and accesses to invalid state generate miss requests to the L2. A read miss does not have to leave the L1 cache in a pending or transient state – since there are no concurrent conflicting accesses (and hence no invalidation requests), the L1 state simply stays invalid for the line until the response comes back.

For a write request to the L1, unlike a conventional protocol, there is no need to get a "permission-to-write" since this permission is implicitly given by the software race-free guarantee. If the cache does not already have the line registered, it must issue a registration request to the L2 to notify that it has the current up-to-date copy of the line and set the registry state appropriately. Since there are no races as show in Figure 2.3, the write can *immediately* set the state of the cache to registered, without waiting for the registration request to complete. Thus, *there is no transient or pending state for writes either.*

The pending read miss and registration requests are simply monitored in the processor's request buffer, just like those of other reads and writes for a single core system. Thus, although the request buffer technically has transient states, these are not visible to external requests – external requests only see stable cache states. The request buffer also ensures that its core's requests to the same location are serialized to respect uniprocessor data dependencies, similar to a single core implementation (e.g., with MSHRs). The memory model requirements are met by ensuring that all pending requests from the core complete by the end of this parallel phase (or at least before the next conflicting access in the next parallel phase).

The L2 transitions are also straightforward except for writebacks which require some care. A read or registration request to data that is invalid or valid at the L2 invokes the obvious response. For a request for data that is registered by an L1, the L2 forwards the request to that L1 and updates its registration id if needed. For a forwarded registration request, the L1 always acknowledges the requestor and invalidates its

	$Read_i$	$Write_i$	$Read_k$	$Register_k$	Response for $Read_i$	Writeback
$Invalid$	Update tag; Read miss to L2; Writeback if needed	Go to $Registered$; Reply to core i; Register request to L2; Write data; Writeback if needed	Nack to core core k	Reply to core k	If tag match, go to $Valid$ and load data; Reply to core i	Ignore
$Valid$	Reply to core i	Go to $Registered$; Reply to core i; Register request to L2	Send data to core k	Go to $Invalid$; Reply to core k	Reply to core i	Ignore
$Registered$	Reply to core i	Reply to core i	Reply to core k	Go to *Invalid*; Reply to core k	Reply to core i	Go to *Valid*; Writeback

(a) **L1 cache of core i. $Read_i$ = read from core i, $Read_k$ = read from another core k (forwarded by the registry).**

	Read miss from core i	Register request from core i	Read response from memory for core i	Writeback from core core i
$Invalid$	Update tag; Read miss to memory; Writeback if needed	Go to $Registered_i$; Reply to core i; Writeback if needed	If tag match, go to $Valid$ and load data; Send data to core i	Reply to core i; Generate reply for pending writeback to core i
$Valid$	Data to core i	Go to $Registered_i$; Reply to core i	X	X
$Registered_j$	Forward to core j; Done	Forward to core j; Done	X	if i==j go to $Valid$ and load data; Reply to core i; Cancel any pending Writeback to core i

(b) **L2 cache**

Table 2.1 DeNovo cache coherence protocol for (a) private L1 and (b) shared L2 caches. Self-invalidation and touched bits are not shown here since these are local operations as described in the text. Request buffers (MSHRs) are not shown since they are similar to single core systems.

own copy. If the copy is already invalid due to a concurrent writeback by the L1, the L1 simply acknowledges the original requestor and the L2 ensures that the writeback is not accepted (by noting that it is not from the current registrant). For a forwarded read request, the L1 supplies the data if it has it. If it no longer has the data (because it issued a concurrent writeback), then it sends a negative acknowledgement (nack) to the original requestor, which simply resends the request to the L2. Because of race-freedom, there cannot be another concurrent write, and so no other concurrent writeback, to the line. Thus, the nack eventually finds the line in the L2, without danger of any deadlock or livelock. The only somewhat less straightforward interaction is when both the L1 and L2 caches want to writeback the same line concurrently, but this race also occurs in uniprocessors.

Conveying and representing regions in hardware: A key research question is how to represent regions in hardware for self-invalidations. Language-level regions are usually much more fine-grain than may be practical to support in hardware. For example, when a parallel loop traverses an array of objects, the compiler may need to identify (a field of) *each object* as being in a distinct region in order to prove the absence of conflicts. For the hardware, however, such fine distinctions would be expensive to maintain. Fortunately, we can coarsen language-level regions to a much smaller set without losing functionality in hardware. The key insight is as follows. We need regions to identify which data could have been written in the current phase for a given core to self-invalidate potentially stale data. It is not important for the self-invalidating core to distinguish which core wrote which data. In the above example, we can thus treat the entire array of objects as one region. So on a self-invalidation instruction, a core self-invalidates all the data in this array (irrespective of whichever core modified it) that is neither read or written by the given core in the given parallel phase.

Alternately, if only a subset of the fields in each object in the above array is written, then this subset aggregated over all the objects collectively forms a hardware region. Thus, just like software regions, hardware regions need not be contiguous in memory – they are essentially an assignment of a color to each heap location (with orders of magnitude fewer colors in hardware than software). Hardware regions are not restricted to arrays either. For example, in a traversal of the spatial tree in an n-body problem, the compiler distinguishes different tree nodes (or subsets of their fields) as separate regions; the hardware can treat the entire tree (or a subset of fields in the entire tree) as an aggregate region. Similarly, hardware regions may also combine field regions from different aggregate objects (e.g., fields from an array and a tree may be combined into one region).

The compiler can easily *summarize* program regions into coarser hardware regions as above and insert appropriate self-invalidation instructions. The only correctness requirement is that the self-invalidated regions must cover all write effects for the phase. For performance, these regions should be as precise as possible. For example, fields that are not accessed or read-only in the phase should not be part of these regions. Similarly, multiple field regions written in a phase may be combined into one hardware region for that phase, but if they are not written together in other phases, they will incur unnecessary invalidations.

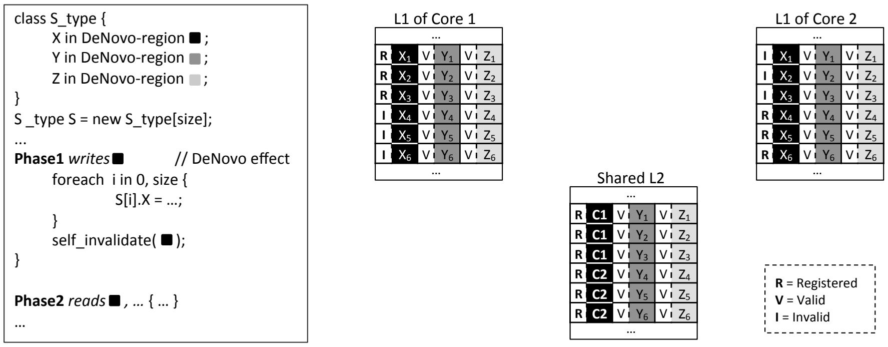
During final code generation, the memory instructions generated can convey the region name of the address being accessed to the hardware; since DPJ regions are parameterizable, the instruction needs to point to a hardware register that is set at runtime (through the compiler) with the actual region number. When the memory instruction is executed, it conveys the region number to the core's cache. A straightforward approach is to store the region number with the accessed data line in the cache. Now a self-invalidate instruction invalidates all data in the cache with the specified regions that is not `touched` or `registered`.

The above implementation requires storing region bits along with data in the L1 cache and matching region numbers for self-invalidation. A more conservative implementation can reduce this overhead. At the beginning of a phase, the compiler conveys to the hardware the set of regions that need to be invalidated in the *next* phase – this set can be conservative, and in the worst case, represent all regions. Additionally, we replace the region bits in the cache with one bit: `keepValid`. indicating that the corresponding data need not be invalidated until the end of the *next* phase. On a miss, the hardware compares the region for the accessed data (as indicated by the memory instruction) and the regions to be invalidated in the next phase. If there is no match, then `keepValid` is set. At the end of the phase, all data not `touched` or `registered` are invalidated and the `touched` bits reset as before. Further, the identities of the `touched` and `keepValid` bits are swapped for the next phase. This technique allows valid data to stay in cache through a phase even if it is not `touched` or `registered` in that phase, without keeping track of regions in the cache. The concept can be extended to more than one such phase by adding more bits and if the compiler can predict the self-invalidation regions for those phases.

Example: Figure 2.4 illustrates the above concepts. Figure 2.4(a) shows a code fragment with parallel phases accessing an array, S, of structs with three fields each, X, Y, and Z. The X (respectively, Y and Z) fields from all array elements form one DeNovo region. The first phase writes the region of X and self-invalidates that region at the end. Figure 2.4(b) shows, for a two core system, the L1 and L2 cache states at the end of Phase 1, assuming each core computed one contiguous half of the array. The computed X fields are `registered` and the others are invalid in the L1's while the L2 shows all X fields registered to the appropriate cores. The example assumes that the caches contained valid copies of B and C from previous computations.

2.2.2 DeNovo with Address/Communication Granularity $>$ Coherence Granularity

To decouple the address/communication and coherence granularity, our key insight is that any data marked `touched` or `registered` can be copied over to any other cache in `valid` state (but not as `touched`). Additionally, for even further optimization (Section 2.6), we make the observation that this transfer can happen without going through the registry/L2 at all (because the registry does not track sharers). Thus, no serialization at a directory is required. When (if) this copy of data is accessed through a demand read, it



```
class S_type {
        X in DeNovo-region ■ ;
        Y in DeNovo-region ▨ ;
        Z in DeNovo-region ░ ;
}
S _type S = new S_type[size];
...
Phase1 writes ■           // DeNovo effect
        foreach  i in 0, size {
                S[i].X = ...;
        }
        self_invalidate( ■ );
}

Phase2 reads ■ , ... { ... }
...
```

(a)

L1 of Core 1

…					
R	X_1	V	Y_1	V	Z_1
R	X_2	V	Y_2	V	Z_2
R	X_3	V	Y_3	V	Z_3
I	X_4	V	Y_4	V	Z_4
I	X_5	V	Y_5	V	Z_5
I	X_6	V	Y_6	V	Z_6
…					

L1 of Core 2

…					
I	X_1	V	Y_1	V	Z_1
I	X_2	V	Y_2	V	Z_2
I	X_3	V	Y_3	V	Z_3
R	X_4	V	Y_4	V	Z_4
R	X_5	V	Y_5	V	Z_5
R	X_6	V	Y_6	V	Z_6
…					

Shared L2

…					
R	C1	V	Y_1	V	Z_1
R	C1	V	Y_2	V	Z_2
R	C1	V	Y_3	V	Z_3
R	C2	V	Y_4	V	Z_4
R	C2	V	Y_5	V	Z_5
R	C2	V	Y_6	V	Z_6
…					

R = Registered
V = Valid
I = Invalid

(b)

Figure 2.4 (a) Code with DeNovo regions and self-invalidations and (b) cache state after phase 1 self-invalidations and direct cache-to-cache communication with flexible granularity at the beginning of phase 2. X_i represents $S[i].X$. Ci in L2 cache means the word is registered with Core i. Initially, all lines in the caches are in `valid` state.

can be immediately marked `touched`. The presence of a demand read means there will be no concurrent write to this data, and so it is indeed correct to read this value (`valid` state) and furthermore, the copy will not need invalidation at the end of the phase (`touched` copy). The above copy does not incur false sharing (nobody loses ownership) and, if the source is the non-home node, it does not require extra hops to a directory.

With the above insight, we can easily enhance the baseline word-based DeNovo protocol from the previous section to operate on a larger communication and address granularity; e.g., a typical cache line size from conventional protocols. However, we still maintain coherence state at the granularity at which the program guarantees data race freedom; e.g., a word. On a demand request, the cache servicing the request can send an entire cache line worth of data, albeit with some of the data marked invalid (those that it does not have as `touched` or `registered`). The requestor then merges the valid words in the response message (that it does not already have `valid` or `registered`) with its copy of the cache line (if it has one), marking all of those words as `valid` (but not `touched`).

Note that if the L2 has a line `valid` in the cache, then an element of that line can be either `valid` (and hence sent to the requestor) or `registered` (and hence not sent). Thus, for the L2, it suffices to keep just one coherence state bit at the finer (e.g., word) granularity with a line-wide valid bit at the line granularity.[1] As before, the id of the registered core is stored in the data array of the registered location.

This is analogous to sector caches – cache space allocation (i.e., address tags) is at the granularity of a

[1] This requires that if a registration request misses in the L2, then the L2 obtain the full line from main memory.

line but there may be some data within the line that is not valid. This combination effectively allows exploiting spatial locality without any false sharing, similar to multiple writer protocols of software distributed shared memory systems [74].

2.3 Flexible Coherence Granularity

Although the applications we studied did not have any data races at word granularity, this is not necessarily true of all applications. Data may be shared at byte granularity, and two cores may incur conflicting concurrent accesses to the same word, but for different bytes. A straightforward implementation would require coherence state at the granularity of a byte, which would be significant storage overhead. [2] Although previous work has suggested using byte based granularity for state bits in other contexts [90], we would like to minimize the overhead.

We focus on the overhead in the L2 cache since it is typically much larger (e.g., 4X to 8X times larger) than the L1. We observe that byte granularity coherence state is needed only if two cores incur conflicting accesses to different bytes in the same word in the same phase. Our approach is to make this an infrequent case, and then handle the case correctly albeit at potentially lower performance.

In disciplined languages, the compiler/runtime can use the region information to allocate tasks to cores so that byte granularity regions are allocated to tasks at word granularities when possible. For cases where the compiler (or programmer) cannot avoid byte granularity data races, we require the compiler to indicate such regions to the hardware. Hardware uses word granularity coherence state. For byte-shared data such as the above, it "clones" the cache line containing it in four places: place i contains the ith byte of each word in the original cache line. If we have at least four way associativity in the L2 cache (usually the case), then we can do the cloning in the same cache set. The tag values for all the clones will be the same but each clone will have a different byte from each word, and each byte will have its own coherence state bit to use (essentially the state bit of the corresponding word in that clone). This allows hardware to pay for coherence state at word granularity while still accommodating byte granularity coherence when needed, albeit with potentially poorer cache utilization in those cases.

Specifically, DeNovo uses three features of these programming models: (1) structured parallel control;

[2] The upcoming C and C++ memory models and the Java memory model do not allow data races at byte granularity; therefore, we also do not consider a coherence granularity lower than that of a byte.

(2) data-race-freedom with guarantees of deterministic execution; and (3) side effects of parallel sections.

2.4 Discussion

In this chapter we used DPJ as an exemplar language that provides all the features of a disciplined programming language that DeNovo can exploit (Section 2.1.1). In Chapter 5 we describe how we apply DeNovo to another language, CUDA, in the context of heterogeneous systems when we introduce a new memory organization called as stash. CUDA provides structured parallel control and partially provides data-race-freedom with deterministic execution. Today's heterogeneous systems require applications written in programming languages such as CUDA and OpenCL to be data-race free even though no such guarantees are provided. DeNovo described in this thesis needs the program to adhere to structured parallel control and deterministic execution. For structured parallel control, the inherent assumption is that a barrier synchronization is the only type of synchronization supported (this barrier can be across a subset of tasks). We assume that the hardware has support for such barrier synchronization.

Yet another difference between today's heterogeneous programming languages and DPJ is the lack of region and effect information in languages like CUDA. DeNovo doesn't necessarily need the region and effect information for its functionality. When such information is not available, DeNovo can be conservative and self-invalidate all the data except that is $touched$ or in $Registered$ state at synchronization points. If the region and effect information is available, DeNovo can perform better by selectively self-invalidating the data at the end of a parallel phase. We quantitatively discuss the benefit of using region and effect information in Section 4.5.

2.5 Delayed Registrations

In Section 2.2.2 we made the communication and the address granularity to be larger than the coherence granularity. However, the registration request granularity is still kept the same as the coherence granularity (e.g., word granularity). If a program has a lot of write requests in a given phase, this implies that multiple registration requests are triggered (one per word) for a given communication/address granularity. This may result in unnecessary increase in the network traffic compared to a single registration request sent for protocols like MESI. So we added a simple optimization to delay registration requests, write combining, that aggregates word granularity registration requests within a communication/address granularity. We have a bounded buffer that holds the delayed registration requests. A buffer entry is drained whenever the buffer gets full or after some threshold time has elapsed (to avoid bursty traffic). The entire buffer is guaranteed to be drained before the next phase begins.

2.6 Protocol Optimizations to Address Communication Inefficiencies

In this section, we extend the DeNovo protocol to add two optimizations. This extension is aimed at both demonstrating the extensibility of the DeNovo protocol and addressing some of the communication inefficiencies in today's memory hierarchies. At a high level, the data that is being communicated can be classified into two broad categories. Both these categories introduce different types of inefficiencies while data is being transferred from one point to another in the memory hierarchy.

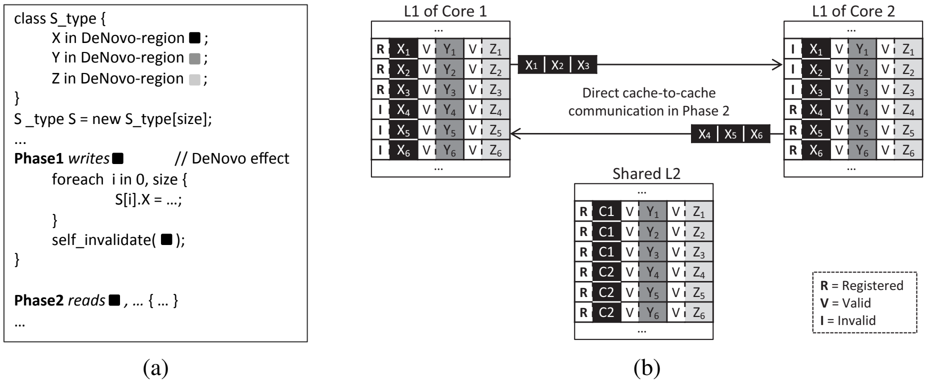
The first category is data that is actually used by the program but could avoid extra hops in the network. For example, if the producer of the data is known in at the time of a request (e.g., an accelerator generates the data and the CPU consumes it) we may avoid an indirection through the directory. Another example is when we have streaming input data that is read only once we may be able to bypass some of the structures like the last level cache because there is no reuse. The second category is data that is never used. This happens because of fixed cache line size data transfers, overwritten data before ever being read, and so on.

The two communication optimizations we describe in this section, specifically, focus on mitigating the inefficiencies at L1, one for each of the two categories mentioned above. We apply these optimizations when evaluating the DeNovo coherence protocol in Section 4. We discuss several directions for future work that aim to address network traffic inefficiencies in other parts of the memory hierarchy in Section 8.2.2.

2.6.1 Direct Transfer

The DeNovo coherence protocol described earlier suffers from the fact that even L1 misses that are eventually serviced by another L1 cache (cache-to-cache transfer) must go through the registry/L2 (directory in conventional protocols), incurring an additional latency due to the indirection.

However, as observed in Section 2.2.2, `touched/registered` data can always be transferred for reading without going through the registry/L2. optimization). Thus, a reader can send read requests directly to another cache that is predicted to have the data. If the prediction is wrong, a Nack is sent (as usual) and



```
class S_type {
        X in DeNovo-region ■ ;
        Y in DeNovo-region ▩ ;
        Z in DeNovo-region □ ;
}
S _type S = new S_type[size];
...
Phase1 writes■          // DeNovo effect
        foreach  i in 0, size {
                S[i].X = ...;
        }
        self_invalidate( ■ );
}

Phase2 reads■ , ... { ... }
...
```

Figure 2.5 Code and cache state from Figure 2.4 with direct cache-to-cache communication and flexible granularity at the beginning of phase 2.

the request reissued as a usual request to the directory. Such a request could be a demand load or it could be a prefetch. Conversely, it could also be a producer-initiated communication or remote write [3, 80]. The prediction could be made in several ways; e.g., through the compiler or through the hardware by keeping track of who serviced the last set of reads to the same region. The key point is that there is no impact on the coherence protocol – no new states, races, or message types. The requestor simply sends the request to a different supplier. This is in sharp contrast to adding such an enhancement to MESI.

This ability essentially allows DeNovo to seamlessly integrate a message passing like interaction within its shared-memory model. Figure 2.5 revisits the example code from Figure 2.4 and shows an interaction between two private caches for a direct cache-to-cache transfer.

2.6.2 Flexible Transfer Granularity

Cache-line based communication transfers data from a set of contiguous addresses, which is ideal for programs with perfect spatial locality and no false sharing. However, it is common for programs to access only a few data elements from each line, resulting in significant waste. This is particularly common in modern object-oriented programming styles where data structures are often in the form of arrays of structs (AoS) rather than structs of arrays (SoA). It is well-known that converting from AoS to SoA form often gives a significant performance boost due to better spatial locality. Unfortunately, manual conversion is tedious, error-prone, and results in code that is much harder to understand and maintain, while automatic (compiler) conversion is impractical except in limited cases because it requires complex whole-program analysis and transformations [52, 71]. We exploit information about regions to reduce such communication waste, without changing the software's view.

We have knowledge of which regions will be accessed in the current phase. Thus, when servicing a remote read request, a cache could send touched or registered data only from such regions (recall these are at field granularity within structures), potentially reducing network bandwidth and power. More generally, the compiler may associate a default prefetch granularity attribute with each region that defines the size of each contiguous region element, other regions in the object likely to be accessed along with this region (along with their offset and size), and the number of such elements to transfer at a time. This information can be kept as a table in hardware which is accessed through the region identifier and an entry provides the above information; we call the table the *communication region table*. The information for the table itself may be partly obtained directly through the programmer, deduced by the compiler, or deduced by a runtime tool. Figure 2.5 shows an example of the use of flexible communication granularity – the caches communicate multiple (non-contiguous) fields of region X rather than the contiguous X, Y, and Z regions that would fall in a conventional cache line. Again, in contrast to MESI, the additional support required for this enhancement in DeNovo does not entail any changes to the coherence protocol states or introduce new protocol races.

This flexible communication granularity coupled with the ability to remove indirection through the registry/L2 (directory) effectively brings the system closer to the efficiency of message passing while still retaining the advantages of a coherent global address space. It combines the benefits of various previously proposed shared-memory techniques such as bulk data transfer, prefetching, and producer-initiated communication, but in a more software-aware fashion that potentially results in a simpler and more effective system.

2.7 Storage Overhead

We next compare the storage overhead of DeNovo to other common directory configurations.

DeNovo overhead: At the L1, DeNovo needs state bits at the word granularity. We have three states and one touched bit (total of 3 bits). We also need region related information. In our applications, we need at most 20 hardware regions – 5 bits. These can be replaced with 1 bit by using the optimization of the keepValid bit discussed in Section 2.2.1. Thus, we need a total of 4 to 8 bits per 32 bits or 64 to 128 bits per L1 cache line. At the L2, we just need one valid and one dirty bit per line (per 64 bytes) and one bit per word, for a total of 18 bits per 64 byte L2 cache line or 3.4%. If we assume L2 cache size of 8X that of L1, then the L1 overhead is 1.56% to 3.12% of the L2 cache size.

In-cache full map directory. We conservatively assume 5 bits for protocol state (assuming more than 16 stable+transient states). This gives 5 bits per 64 byte cache line at the L1. With full map directories, each L2 line needs a bit per core for the sharer list. This implies that DeNovo overhead for just the L2 is better for more than a 13 core system. If the L2 cache size is 8X that of L1, then the total L1+L2 overhead of DeNovo is better at greater than about 21 (with `keepValid`) to 30 cores.

Duplicate tag directories. L1 tags can be duplicated at the L2 to reduce directory overhead. However, this requires a very high associative lookup; e.g., 64 cores with 4 way L1 requires a 256 way associative lookup. As discussed in [141], this design is not scalable to even low tens of cores system.

Tagless directories and sparse directories. The tagless directories work uses Bloom filter based directory organization [141]. Their directory storage requirement appears to be about 3% to over 5% of L1 storage for core counts ranging from 64 to 1K cores. This does not include any coherence state overhead which we include in our calculation for DeNovo above. Further, this organization is lossy in that larger core counts require extra invalidations and protocol complexity.

Many sparse directory organizations have been proposed that can drastically cut directory overhead at the cost of sharer list precision, and so come at a significant performance cost especially at higher core counts [141].

2.8 Summary

We introduced DeNovo, a software-hardware co-designed coherence protocol, that aims to address several inefficiencies with today's traditional directory-based protocols. DeNovo proposes to address these inefficiencies by exploiting the properties of the emerging disciplined programming models. Specifically, DeNovo uses three features of these programming models: (1) structured parallel control; (2) data-race-freedom with guarantees of deterministic execution; and (3) side effects of parallel sections.

One of the benefits of DeNovo is that it is simple and thus, makes it easy to extend the protocol for optimizations. In this chapter we extended DeNovo by adding two optimizations to the protocol addressing communication inefficiencies. We showed that neither of the two optimizations required addition of new protocol states or races. We go a step further and validate the simplicity of the DeNovo protocol by formally verifying it and comparing this effort against that of verifying a state-of-the-art and publicly available implementation of a traditional protocol (MESI). The next chapter (Chapter 3) provides more details on this effort and its findings. In addition to evaluating the complexity of the DeNovo protocol, we also provide performance evaluations of the protocol including the communication optimizations for several applications in Chapter 4.

CHAPTER 3

COMPLEXITY ANALYSIS OF THE DENOVO PROTOCOL

One of the benefits of the DeNovo coherence protocol introduced in Chapter 2 is its simplicity. Thus DeNovo is expected to incur a reduced verification effort compared to the traditional hardware protocols. In this chapter, we describe our efforts to verify the DeNovo protocol (this work appears in [78]) using the Murφ model checking tool (version 3.1, slightly modified to exploit 64-bit machines) [54, 69, 105]. Although more advanced verification techniques exist, we chose Murφ for its easy-to-use interface and robustness. Murφ has also been the tool of choice for many hardware cache related studies [5, 34, 109, 110, 144].

This verification effort has two goals - (1) verify the correctness of the DeNovo coherence protocol; and (2) compare our experience of verifying the DeNovo protocol with that of a state-of-the-art, mature, and publicly available protocol and validate the simplicity of the DeNovo protocol. For the latter goal, we chose the MESI protocol implemented in the Wisconsin GEMS simulation suite (version 2.1.1) [94]. It is difficult to define a metric to quantify the relative verification complexity of coherence protocols; nevertheless, our results demonstrate that hardware-software co-designed approaches like DeNovo can lead to much simpler protocols than conventional hardware cache coherence (while providing an easy programming model, extensibility, and competitive or better performance).

We do not quantify the software complexity in this chapter; however, our software philosophy and DPJ are motivated entirely by the goal of *reducing software complexity*. Even today, the C++ [31] and Java [93] memory models do not provide any reasonable semantics for data races; therefore, a data race in these programs is a bug and imposes significant verification complexity. In contrast, DPJ provides strong safety guarantees of data-race-freedom and determinism-by-default. Programmers can reason about deterministic programs as if they were sequential. There is certainly an additional up-front burden of writing region and effect annotations in DPJ; however, arguably this burden is mitigated by the lower debugging and testing time afforded by deterministic-by-default semantics. There is also ongoing work on partly automating the insertion of these annotations [134].

3.1 Modeling for Protocol Verification

We use the Murφ model checking tool [54, 69, 105] to verify the simple word based protocols (equal address, communication and coherence granularity as explained in Section 2.2.1) of DeNovo and MESI. We derived the MESI model from the GEMS implementation [94]. We derived the DeNovo model from our own implementation. To keep the number of states explored (by Murφ) tractable, as is common practice, we used a single address, single region (only for DeNovo), two data values, and two cores. We modeled private $L1$ caches, a unified $L2$, an in-cache directory (for MESI) and an unordered full network with separate request and reply links. Both models allow only one request per $L1$ in the rest of the memory hierarchy. As we modeled only one address, we modeled replacements as unconditional events that can be triggered at any time. To enable interactions across multiple parallel phases (cross-phase) in both the models, we introduced the notion of a phase boundary by modeling it as a sense reversing barrier. Finally, we modeled the data-race-free guarantee for DeNovo by limiting conflicting accesses. We explain each of these attributes in detail below.

3.1.1 Abstract Model

To reduce the amount of time and memory used in verification, we modeled the processors, addresses, data values, and regions as scalarsets [105], a datatype in Murφ, which takes advantage of the symmetry in these entitites while exploring the reachable states. A processor is modeled as an array of cache entries consisting of $L1$ state information along with protocol specific fields like the region field and the $touched$ bit for DeNovo. $L1$ state is one of 3 possible states for DeNovo or one of 11 possible states for MESI. Similarly, $L2$ is also modeled as an array of cache entries, each with $L2$ state information, $dirty$ bit, and other protocol specific details like sharer lists for MESI. $L2$ state is one of 3 possible states for DeNovo or one of 18 possible states for MESI. Memory is modeled as an array of addresses storing data values.

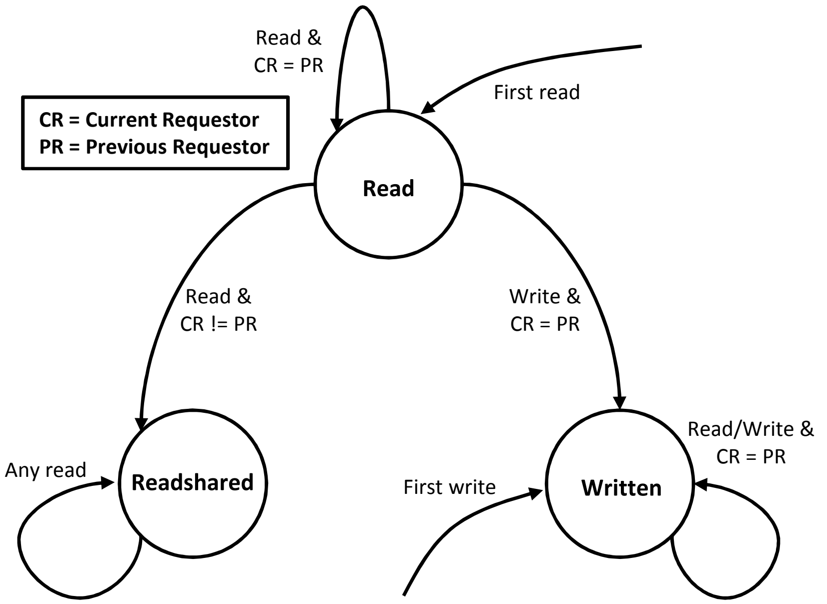


Figure 3.1 State transitions for AccessStatus data structure in a given phase. An access for which there is no transition cannot occur and is dropped.

Data-race-free Guarantee for DeNovo

To model the data-race-free guarantee from software for DeNovo, we used an additional data structure called $AccessStatus$. As shown in Figure 3.1, this data structure maintains the current status ($read$, $readshared$, or $written$) and the core id of the last requestor for every address in the model. The current status and the last requestor determine the reads and writes that cannot occur in a data-race-free program and are thus disallowed in the model.

On any read, if it is the first access to this address in this phase, then $status$ is set to $read$. If $status$ is already set to $read$ and the requesting core is not the same as the last requestor, then $status$ is set to $readshared$. If $status$ is $readshared$, then it stays the same on the read. If $status$ is $written$ and the requesting core is the same as the last requestor it stays as $written$. On the other hand, if the requesting core is not the same as the last requestor, then this access is not generated in the model since it violates the data-race-freedom guarantee.

Similarly, on any write, if it is the first access to this address or if the requesting core is the same as the last requestor, then $status$ is set to $write$. If $status$ is either $readshared$ or the requesting core is not the same as the last requestor, then this access is not generated to adhere to the data-race-free guarantee.

The $AccessStatus$ data structure is reset for all the addresses at the end of a phase.

Cross-phase Interactions

We modeled the end of a parallel phase (and the start of the next phase) using a sense-reversing barrier implementation [100]. This event (end-of-phase) can be triggered at any time; i.e., with no condition. The occurrence of end-of-phase is captured by a flag, $releaseflag$. This event occurs per core and stalls the core from issuing any more memory requests until (1) all the pending requests of this core are completed; i.e., the $L1$ request buffer is empty and (2) all other cores reach the barrier. The completion of end-of-phase is indicated by resetting the $releaseflag$ flag. Figure 3.2 shows the Murφ code for end-of-phase implementation for the DeNovo protocol. The $spinwaiting$ flag indicates that the current core is waiting for other cores to reach the barrier. When a core enters the barrier for the first time, the local sense of the barrier ($localsense$) is reversed indicating entering a new barrier, barrier count ($barcount$) is updated, and the $spinwaiting$ flag is set. If it is the last one to enter the barrier, the core also notifies all other cores about the end of barrier by assigning $barrier$ its $localsense$. It also resets the $barcount$ and $releaseflag$. Once a core reaches the barrier, we also modeled self-invalidations and unsetting $touched$ bits for DeNovo. The code for MESI is similar except for DeNovo-specific operations like self-invalidation and unsetting $touched$ bits.

3.1.2 Invariants

This section discusses the invariants we checked to verify the MESI and DeNovo protocols. The MESI invariants are based on prior work in verification of cache coherence protocols [54, 97]. The DeNovo invariants are analogous as further described below. (Adding more invariants does not affect the verification time appreciably because the number of system states explored is still the same.)

MESI Invariants

We used five invariants to verify the MESI protocol [54, 97].

Empty sharer list in Invalid state. This invariant asserts that the sharer list is empty when $L2$ transitions to $Invalid$ state, and ensures that there are no $L1$s sharing the line after $L2$ replaces the line.

```
/* barrier implementation */
Ruleset p:Proc Do
  Rule "Advance release barrier"
    releaseflag[p] & L1_isRequestBufferEmpty(p)
  ==>
  Begin
    advanceRelease(p);
  Endrule;
Endruleset;
```

(a) Murφ barrier rule.

```
Procedure advanceRelease(pid:Proc);
Begin
  If !spinwaiting[pid] then
    /* not spinwaiting => executing this
       barrier code for first time */
    /* a sense reversing barrier */
    localsense[pid]    := !localsense[pid];
    /* no need for a lock here as
       Murφ's procedures are atomic */
    barcount           := barcount + 1;
    If barcount = ProcCount then
      barcount         := 0;
      barrier          := localsense[pid];
      releaseflag[pid] := false;
      /* self-invalidate and unset touched
         bits after release is done */
      self_invalidate(pid);
      unset_touched(pid);
      /* the last proc to set the barcount
         to 0 also unsets the AccessStatus */
      unsetAccessStatus();
    Else
      spinwaiting[pid] := true;
    Endif;
  Else
    If barrier = localsense[pid] then
      releaseflag[pid] := false;
      spinwaiting[pid] := false;
      /* self-invalidate and unset
         touched bits after release is done */
      self_invalidate(pid);
      unset_touched(pid);
    Endif;
  Endif;
EndProcedure;
```

(b) Murφ procedure for implementing a sense-reversing barrier.

Figure 3.2 Murφ code for end-of-phase implementation for DeNovo as a sense-reversing barrier. (a) Rule that gets triggered when inside the barrier indicated by $releaseflag$ and empty $L1$ request buffer (no outstanding requests) and (b) implementation of the sense-reversing barrier including calls to end-of-phase operations like $self$-$invalidation$ instruction and unsetting of $touched$ bits.

Empty sharer list in Modified state. This invariant asserts that the sharer list is empty when $L2$ transitions to $Modified$ state.

Only one modifiable or exclusive cache copy. This invariant checks that there is only one cache copy in either $Modified$ or $Exclusive$ state. It is also a violation for a cache line to be in both these states at the same time.

Data value consistency at $L1$. When $L1$ is in $Shared$ state and $L2$ is also in $Shared$ state, the data values should be the same at both $L1$ and $L2$. Indirectly, this invariant also makes sure that all the $L1$s have the same data value when in $Shared$ state.

Data value consistency at $L2$. This invariant checks that when $L2$ is in $Shared$ state and $dirty$ bit is not set, $L2$'s data value should be the same as at memory.

DeNovo Invariants

We modeled six invariants for the DeNovo protocol. As there is no sharer list maintained in the DeNovo protocol, we do not check for the first two invariants of the MESI protocol. The first three invariants of the DeNovo protocol are similar to the last three invariants of the MESI protocol. The last three invariants of the DeNovo protocol are checks on the $touched$ bit functionality.

Only one modifiable cache copy. There cannot be two modifiable $L1$ cache copies in the system at the same time. This invariant checks that there are never two $L1$ caches in $Registered$ state for the same line at the same time.

Data value consistency at $L1$. This invariant has two parts: (i) If $L1$ is in $Valid$ state and $touched$ bit is set (value is read in this phase) and $L2$ is also in $Valid$ state, then the data values should be the same at both $L1$ and $L2$. (ii) If $L1$ is in $Valid$ state and $touched$ bit is set and some other $L1$ is in $Registered$ state,[1] the data values should match.

Data value consistency at $L2$. This invariant checks that when $L2$ is in $Valid$ state and $dirty$ bit is not set, $L2$'s data value should be the same as at memory.

[1] This is possible in DeNovo because registration at the other $L1$ may have happened in a previous parallel phase.

Touched bit on a write. On a write, this invariant checks that no other cache has the $touched$ bit set to true. This verifies that the $touched$ bit is implemented correctly.

Touched bit on a read. Similar to the above, on a read, this invariant checks that the only cache lines that can have the $touched$ bit set to true (for cores other than the requestor) are the ones in $Valid$ state.

Unsetting $touched$ bits. Finally, this invariant checks that all the $touched$ bits are set to false at the end of the phase.

3.2 Results

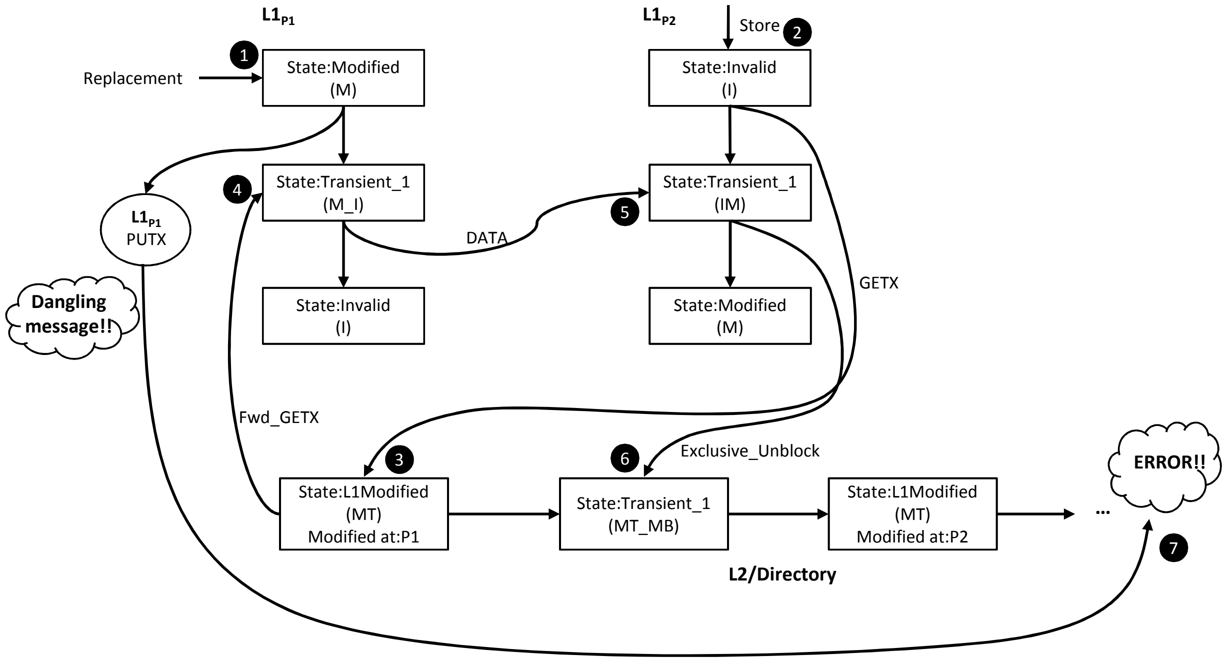


Figure 3.3 MESI Bug 1 showing a race between an $L1$ writeback and a remote write request.

Our findings from applying Murφ to the GEMS MESI protocol were surprising. We found six bugs in the protocol (including two deadlock scenarios[2]), even though it is a mature protocol (released in 2007) used by a large number of architecture researchers. More significantly, some of these bugs involved subtle races and took several days to debug and fix. We contacted the developers of the GEMS simulation team with our bug findings in 2011. They had seen one of the six bugs, but were surprised at the other bugs.

[2] A deadlock occurs when all the entities in the system (all $L1$s and $L2$) stop making any forward progress. Murφ checks for deadlock by default.

Some of these bugs were also present in the GEM5 simulator [25], an extension to the GEMS simulator to incorporate the M5 CPU core simulator, at that time. After we showed our fixes, the GEMS group fixed the bugs and released new patches. These fixes needed the addition of multiple new state transitions and extra buffer space for stalling requests in the protocol.

Despite DeNovo's immaturity, we found only three bugs in the implementation. Furthermore, these bugs were simple to fix and turned out to be mistakes in translating the high level description of the protocol into the implementation (i.e., their solutions were already present in our internal high level description of the protocol).

Each of the bugs found in MESI and DeNovo is described in detail next. In all the descriptions below, we consider a single address. $L1_{P1}$, $L1_{P2}$, and $L2$ indicate the cache lines corresponding to the above address in core $P1$, core $P2$, and $L2$ respectively. As mentioned in Section 3.1, we assume an in-cache directory at $L2$ and hence we use the words $directory$ and $L2$ interchangeably.

3.2.1 MESI Bugs

We first discuss the six bugs found in the MESI protocol. We list them in decreasing order of their complexity and the amount of change to the code required to fix them. [3]

Bug 1. The first bug is caused by a race between an $L1$ writeback and a write request by some other $L1$. Figure 3.3 shows the events that lead to this bug. Let us assume that initially $L1_{P1}$ is in $Modified$ state, $L1_{P2}$ is in $Invalid$ state, and $L2$ records that the cache entry is modified in $L1_{P1}$. Then $L1_{P1}$ issues a replacement (event 1 in Figure 3.3) triggering a writeback ($PUTX$) and transitions to a transient state waiting for an acknowledgement to this writeback request. Meanwhile, $L1_{P2}$ issues a write request (event 2) triggering $GETX$ to $L2$. $L2$ first receives $GETX$ from $L1_{P2}$ (event 3). It forwards the request to $L1_{P1}$ and waits for an acknowledgement from $L1_{P2}$. $L1_{P1}$, on receiving the $GETX$ request (event 4), forwards the data to $L1_{P2}$ and transitions to $Invalid$ state. Then $L1_{P2}$, on receiving the data from $L1_{P1}$ (event 5) transitions to $Modified$ state and unblocks the directory which in turn records that the cache entry is now modified in $L1_{P2}$. But the writeback ($PUTX$) sent by $L1_{P1}$ is still in the network and it can reach the directory at any time as we have an unordered network (event 7), causing an error. For example, suppose $L1_{P1}$ later services a write request invalidating $L1_{P2}$ and the directory is appropriately updated (not shown

[3] We confirmed the fixes for the MESI bugs in a personal email exchange withone of the developers of the GEMS simulation suite.

in the figure). $L1_{P1}$'s writeback ($PUTX$) then reaches the directory, which is clearly an error. The bug was found when the writeback acknowledgement from $L2$ reached $L1_{P1}$ triggering a "missing transition" failure ($L1_{P1}$ does not expect a writeback acknowledgement in $Modified$ state).

We solved this problem by not transitioning $L1_{P1}$ to $Invalid$ state on receiving $L1_{P2}$'s $GETX$ request. $L1_{P1}$ now sends $DATA$ to $L1_{P2}$ like before, but continues to stay in the transient state, M_I. The write request from $L1_{P1}$, which triggered the bug in the previous example, is now kept pending as $L1_{P1}$ is in a transient state. We also added a transition at the $L2$ to send a writeback acknowledgement when the requester is not the owner in the directory's record. $L1_{P1}$ transitions to $Invalid$ state on receiving the writeback acknowledgement from $L2$. With this, there is no longer a dangling $PUTX$ in the network and the problem is solved. The trace for this bug involved multiple writes to the same memory location in a parallel phase. This scenario does not arise in DeNovo as the software guarantees data-race-freedom.

Bug 2. The second bug is similar to the first bug except that it is caused by a race between an $L1$ writeback and a read request by some other $L1$.

The first two bugs were the most complex to understand and fix. Most of the time was spent in discovering the root cause of the bugs and developing a solution in an already complex protocol. The solutions to these bugs required adding two new cache events and eight new transitions to the protocol.

Bug 3. The third bug is caused by an unhandled protocol race between $L2$ and $L1$ replacements. To begin with, $L1_{P1}$ is in $Exclusive$ state and $L2$ records that $P1$ is the exclusive owner. Then both $L2$ and $L1$ replace the lines simultaneously, triggering invalidation and writeback messages respectively. $L1_{P1}$, on receiving the invalidation message, transitions to $Invalid$ state and sends its data to $L2$. On receiving this data, $L2$ completes the rest of the steps for the replacement. In the end, both $L1$ and $L2$ have transitioned to $Invalid$ states, but the initial writeback message from $L1$ is still in the network and this is incorrect. The bug was found when the writeback acknowledgement (issued by $L2$ on receiving the dangling writeback message) reaches $L1_{P1}$ when it is not expecting one and hence triggers a "missing transition" error.

This bug can be fixed by not sending the data when $L1$ receives an invalidation message and by treating the invalidation message itself as the acknowledgement for $L1$'s earlier writeback message. Also, the $L1$ writeback message is treated as the data response for the invalidation message at $L2$. The fix required adding four new transitions to the protocol.

Bug 4. The fourth bug results in a deadlock situation. It is caused by an incorrectly handled protocol race between an $Exclusive\_unblock$ (response sent to unblock $L2$ on receiving an exclusive access) and an $L1$ writeback issued by the same $L1$ (issued after sending $Exclusive\_unblock$). Initially, $L2$ is waiting for an $Exclusive\_unblock$ in a transient state transitioned from $Invalid$ state. In this transient state, when $L2$ receives an $L1$ writeback, it checks whether this writeback came from the current owner or not. The owner information is updated at $L2$ on receiving the $Exclusive\_unblock$ message. Here, $L1$ writeback (racing with $Exclusive\_unblock$ from the same $L1$) reached $L2$ first and $L2$ incorrectly discarded the $L1$ writeback as the owner information at $L2$ did not match the sender of the $L1$ writeback. This incorrect discarding of the $L1$ writeback results in a deadlock.

This bug can be fixed by holding the $L1$ writeback to be serviced until $Exclusive\_unblock$ is received by $L2$. This requires adding a new transition and additional buffering to hold the stalled request to the protocol.

Bug 5. The fifth bug is similar to the fourth (race between $Exclusive\_unblock$ and $L1$ writeback), but instead $L2$ is initially in $Shared$ state. The fix for this bug required adding two new transitions and additional buffering to hold the stalled requests to the protocol.

Bug 6. The last bug results in a deadlock scenario due to an incorrect transition by $L2$ on a clean replacement. It transitions to a transient state awaiting an acknowledgement from memory even though the transition did not trigger any writeback. The fix was simple and required transitioning to $Invalid$ state instead.

3.2.2 DeNovo Bugs

We next discuss the three bugs found in the DeNovo protocol. The first bug is a performance bug and the last two are correctness bugs, both of which are caused by races related to writebacks.

Bug 1. The first of the three bugs found was caused by not unsetting the dirty bit on replacement of a dirty $L2$ cache line. Assume that $L2$ is initially in $Valid$ state and the $dirty$ bit is set to true. Then on $L2$ replacement, it transitions to $Invalid$ state and writes back data to memory. But the $dirty$ bit is mistakenly not unset. This bug was found when Murφ tried to replace the line in $Invalid$ state as the $dirty$ bit was set to true (the model triggers a replacement event by only checking the $dirty$ bit). The model, legitimately, did not have an action specified for a replacement event in the $Invalid$ state, thus resulting in a "missing transition" error. However, the actual implementation did have an action (incorrectly) that triggered unnecessary writebacks

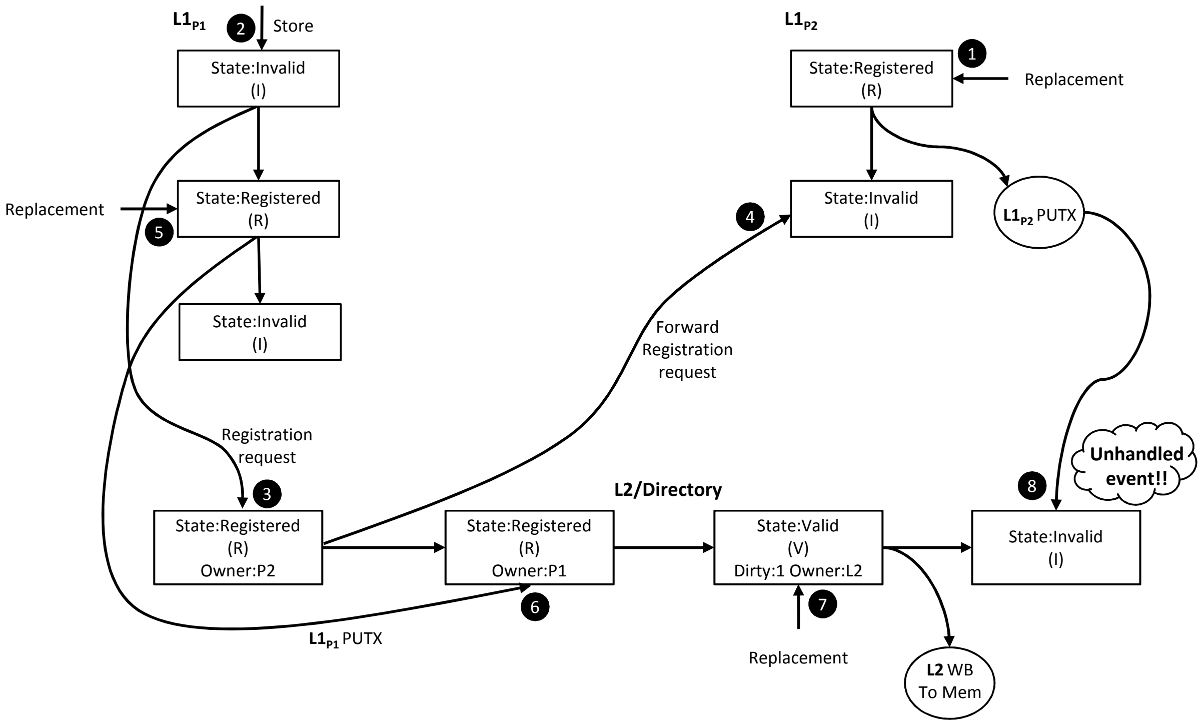


Figure 3.4 DeNovo Bug 3 showing a race between replacements at both the L1s and the L2. This figure doesn't show the request buffer entries for L2 and for writeback entries at L1.

to memory which should be silent replacements instead. This turned out to be a rare case to hit in the simulation runs.

Bug 2. This occurs because an $L2$ initiated writeback and future requests to the same cache line are not serialized. Initially, $L1_{P1}$ is in $Registered$ state and $L2$ knows $P1$ as the registrant. On replacing the line, $L2$ sends a writeback request to $L1$. $L1$ replies to this writeback request by sending the data to $L2$ and transitions to $Valid$ state.[4] Then on receiving the writeback from $L1$, $L2$ sends an acknowledgement to $L1$ and in parallel sends a writeback to memory and waits for an acknowledgement. Meanwhile, let us assume that $L1$ issued a registration request (on receiving a store request) and successfully registers itself with $L2$. At this point, yet another $L2$ replacement was triggered, finally leading to multiple writebacks to memory in flight. This is incorrect because the writebacks can be serviced out of order. Murφ found this bug when an assertion failed inside the implementation of $L2$'s request buffer.

The real source of this bug is allowing $L1$ registration to be serviced at $L2$ while a writeback to memory is pending. The fix involves serializing requests to the same location at $L2$ – in this case, the $L1$ registration

[4] In DeNovo, the L2 cache is inclusive of only the registered lines in any L1. Hence it is possible for $L1$ to transition from $Registered$ to $Valid$ on receiving a writeback request from $L2$.

request behind the writeback to memory. This was already present in our high level specification, but was missed in the actual protocol implementation. It did not involve adding any new states or transitions to the protocol.

Bug 3. The last bug is due to a protocol race where both the $L1$s and the $L2$ replace the line. This bug involves both cores and cross-phase interactions. The events that lead to the bug are shown in Figure 3.4. At the beginning of the phase, let us assume that $L1_{P1}$ is in $Invalid$ state and $L1_{P2}$ is in $Registered$ state (from the previous phase). $L1_{P2}$ replaces the line (event 1 in Figure 3.4) and issues a writeback (PUTX) to $L2$. While this writeback is in flight, $L1_{P1}$ successfully registers itself with $L2$ (events 2-4) ($L2$ redirects the request to $L1_{P2}$ as it is the current registrant). This is followed by a replacement by $L1_{P1}$ (event 5), thus triggering another writeback (PUTX) to $L2$. $L2$ first receives the writeback from $L1_{P1}$ (event 6) and responds by sending an acknowledgement and transitioning to $Valid$ state while setting the $dirty$ bit to true. Now, $L2$ also replaces the line (event 7) transitioning to $Invalid$ state and issues a writeback to memory. But the writeback from $L1_{P2}$ is still in flight. This writeback now reaches $L2$ (event 8) while in $Invalid$ state (because we model an unordered network). The implementation did not handle this case, and resulted in a "missing transition" failure. This bug can be easily fixed by adding a transition to send an acknowledgement to $L1_{P2}$'s writeback, without the need for triggering any actions at $L2$.

3.2.3 Analysis

The bugs described above for both MESI and DeNovo show that cache line replacements and writebacks, when interacting with other cache events, cause subtle races and add to the complexity of cache coherence protocols. Fixes to bugs in the MESI protocol needed adding new events and several new transitions. On the other hand, fixing bugs in the DeNovo protocol was relatively easy since it lacks transient states even for races related to writebacks.

3.2.4 Verification Time

After fixing all the bugs, we ran the models for both MESI and DeNovo on Murφ as described in Section 3.1. The model for MESI explores 1,257,500 states in 173 seconds whereas the model for DeNovo explores 85,012 states in 8.66 seconds. These are the number of distinct system states exhaustively explored by the model checking tool. The state space and runtime both grow significantly when we increase the parameters in the verification model. For example, when we modeled two addresses, we were able to finish running DeNovo without any bugs being reported but we ran out of system memory (32 GB) for MESI. This indicates (1) the simplicity and reduced verification overhead for DeNovo compared to MESI, and (2) the need for more scalable tools amenable to non-experts to deal with more conventional hardware coherence protocols in a more comprehensive way.

3.3 Summary

We described our efforts to verify the DeNovo protocol introduced in Chapter 2. We provided details of our system modeled using the Murφ model checking tool. To evaluate the simplicity of the DeNovo protocol we compare our verification efforts with that of a publicly available, state-of-the-art implementation of the MESI protocol. Surprisingly, we found that after four years of extensive use in the architecture community, the MESI protocol implementation still had several bugs. These bugs were hard to diagnose and fix, requiring new state transitions. In contrast, verifying a far less mature, hardware-software co-designed protocol, DeNovo, revealed fewer bugs that were much easier to fix. After the bug fixes, we found that MESI explored 15X more states and took 20X longer to model check compared to DeNovo. Although it is difficult to define a single metric to quantify the relative complexity of protocols or to generalize from two design points, our results indicate that the DeNovo protocol which is hardware-software co-designed provides a simpler alternative to traditional hardware protocols.

CHAPTER 4

PERFORMANCE EVALUATION OF THE DENOVO PROTOCOL

In this chapter we evaluate the DeNovo coherence protocol that addresses several inefficiencies in today's directory-based coherence protocols. We also evaluate the performance and energy implications of the two protocol optimizations that are applied to DeNovo to address some of the communication inefficiencies.

4.1 Simulation Environment

Our simulation environment consists of the Simics full-system functional simulator that drives the Wisconsin GEMS memory timing simulator [94] which implements the simulated protocols. We also use the Princeton Garnet [8] interconnection network simulator to accurately model network traffic. We chose not to employ a detailed core timing model due to an already excessive simulation time. Instead, we assume a simple, single-issue, in-order core with blocking loads and 1 CPI for all non-memory instructions. We also assume 1 CPI for all instructions executed in the OS and in synchronization constructs.

We used the implementation of MESI that was shipped with GEMS [94] for our comparisons. The original implementation did not support non-blocking stores. Since stores are non-blocking in DeNovo, we modified the MESI implementation to support non-blocking stores for a fair comparison. Our tests show that MESI with non-blocking stores outperforms the original MESI by 28% to 50% (for different applications).

Table 4.1 summarizes the key common parameters of our simulated systems. Each core has a 128KB private L1 Dcache (we do not model an Icache). L2 cache is shared and banked (512KB per core). The latencies in Table 4.1 are chosen to be similar to those of Nehalem [62], and then adjusted to take some properties of the simulated processor (in-order core, two-level cache) into account.

Processor Parameters	
Frequency	2GHz
Number of cores	64
Memory Hierarchy Parameters	
L1 (Data cache)	128KB
L2 (16 banks, NUCA)	32MB
Memory	4GB, 4 on-chip controllers
L1 hit latency	1 cycle
L2 hit latency	29 $\sim$ 61 cycles
Remote L1 hit latency	35 $\sim$ 83 cycles
Memory latency	197 $\sim$ 261 cycles

Table 4.1 Parameters of the simulated processor.

4.2 Simulated Protocols

We compared the following 9 systems:

MESI word (MW) and line (ML): MESI with single-word (4 byte) and 64-byte cache lines, respectively.

DeNovo word (DW) and line (DL): DeNovo with single-word (Section 2) and 64-byte cache lines, respectively.

We charge network bandwidth for only the valid part of the cache line plus the valid-word bit vector.

DL with write-combining (DLC): Line-based DeNovo with the write-combining optimization described in 2.5. We have a bounded buffer of 256 entries per L1.

DLC with direct cache-to-cache transfer (DLCD): DLC with direct cache-to-cache transfer. We use oracular knowledge to determine the cache that has the data. This provides an upper-bound on achievable performance improvement.

DLC with flexible communication granularity (DLCF): DLC with flexible communication granularity. Here, on a demand load, the communication region table is indexed by the region of the demand load to obtain the set of addresses that are associated with that load, referred to as the *communication space*. We fix the maximum data communicated to be 64 bytes for DF. If the communication space is smaller than 64 bytes, then we choose the rest of the words from the 64-byte cache line containing the demand load address. We optimistically do not charge any additional cycles for determining the communication space and gathering/scattering that data.

DLC and DW with both direct cache-to-cache transfer and flexible communication granularity (DLCDF and DWDF respectively): Line-based (with write-combining) and word-based DeNovo with the above two optimizations, direct cache-to-cache transfer and flexible communication granularity, combined in the obvious way.

We do not show word based DeNovo augmented with just direct cache-to-cache transfer or just flexible communication granularity because the results were as expected and did not lend new insights, and the DeNovo word based implementations have too much tag overhead compared to the line based implementations.

4.3 Conveying Regions and Communication Space

4.3.1 Regions for Self-invalidation

In a real system, the compiler would convey the region of a data through memory instructions (Chapter 2). For this study, we created an API to manually instrument the program to convey this information for every allocated object. This information is maintained in a table in the simulator. At every load or store, the table is queried to find the region for that address (which is then stored with the data in the L1 cache).

4.3.2 Self-invalidation

This API call specifies a region and triggers invalidations for the data in the cache associated with this given region that is not `touched` or `registered`.

4.3.3 Conveying Communication Space

To convey communication granularity information, we use a special API call that controls the communication region table of the simulator. On a demand load, the table is accessed to determine the communication space of the requested word. In an AoS program, this set can be simply defined by specifying 1) what object fields, and 2) how many objects to include in the set. For our benchmarks, we manually insert these API calls.

4.3.4 Hardware Overheads

For the applications studied in this paper (see below), the total number of regions ranged from 2 to about 20. These could be coalesced by the compiler, but we did not explore that here. So for these applications, the maximum number of registers needed in the hardware to store the region IDs is 20. Each of these registers need to be at least 5 bit wide to be able to point to any of these 20 regions. The storage overhead at the L1 and the L2 is described in Section 2.7. In our simulations, we do not use the `keepValid` bit. Instead, we explicitly store the region IDs per word. Finally, the communication region table needs to be as big as the maximum number of regions (20 entries for our applications). Each entry in this table stores the prefetch information provided by the compiler for the particular region.

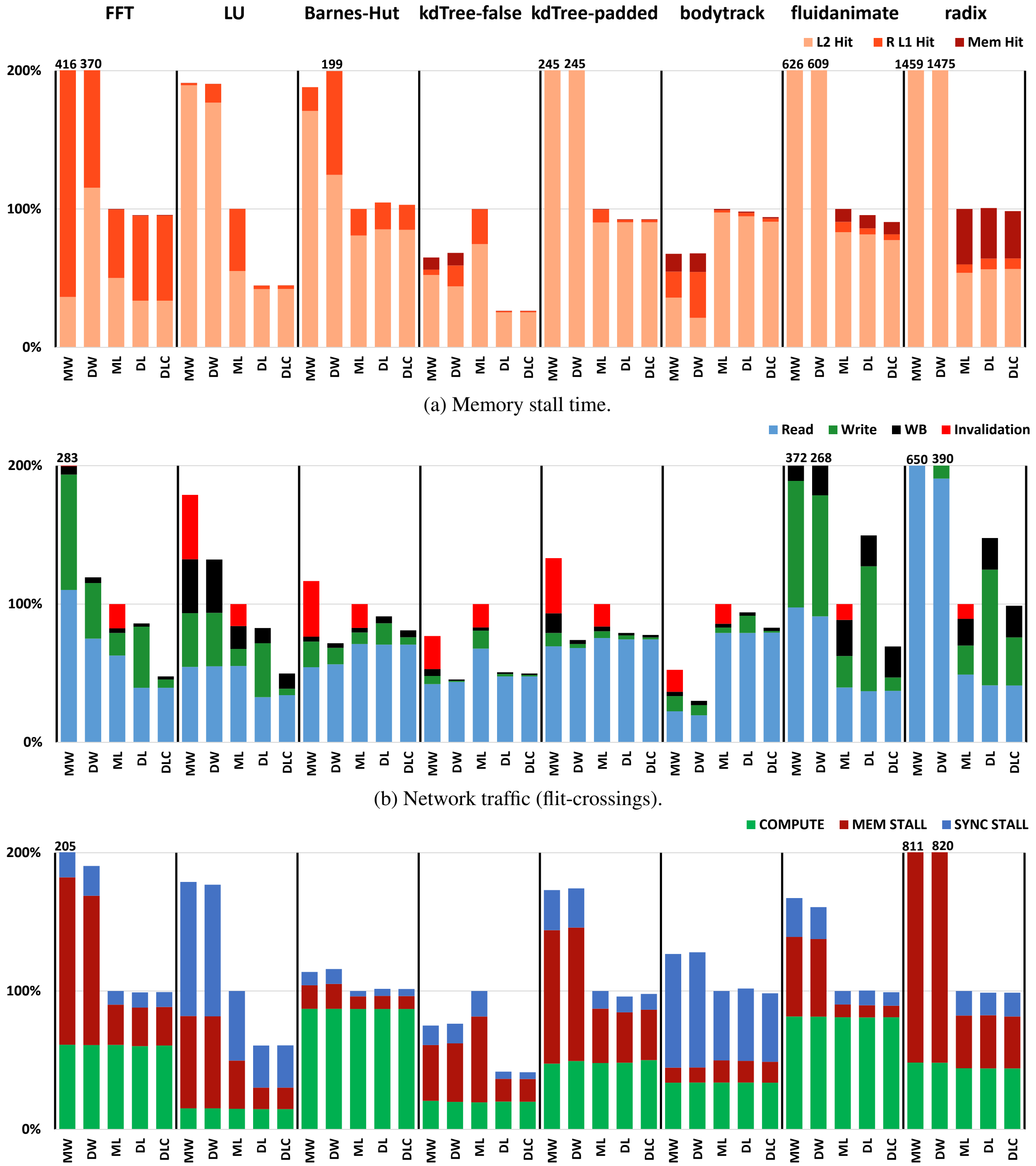
4.4 Workloads

We use seven benchmarks to evaluate the effectiveness of DeNovo features for a range of dense-array, array-of-struct, and irregular pointer-based applications. FFT (with input size m=16), LU (with 512x512 array and 16-byte blocks), Radix (with 4M integers and 1024 radix), and Barnes-Hut (16K particles) are from the SPLASH-2 benchmark suite [139]. kdTree [44] is a program for construction of k-D trees which are well studied acceleration data structures for ray tracing in the increasingly important area of graphics and visualization. We run it with the well known bunny input. We use two versions of kdTree: kdTree-false which has false sharing in an auxiliary data structure and kdTree-padded which uses padding to eliminate this false sharing. We use these two versions to analyze the effect of application-level false sharing on the DeNovo protocols. We also use fluidanimate (with simmedium input) and bodytrack (with simsmall input) from the PARSEC benchmark suite [23]. To fit into the fork-join programming model, fluidanimate was modified to use the ghost cell pattern instead of mutexes, and radix was modified to perform a parallel prefix with barriers instead of condition variables. For bodytrack, we use its pthread version unmodified.

For all the applications above, we manually inserted the API calls for specifying regions, self-invalidations, and conveying communication spaces. These API calls are intercepted by Simics and triggered necessary actions in the simulator. The applications we used are written in C++ and we used g++ (version 4.5.2) to generate the binaries.

4.5 Results

Effectiveness of regions and touched bits: To evaluate the effectiveness of regions and touched bits, we ran DL without them. This resulted in all the valid words in the cache being invalidated by the self-invalidation instruction. Our results show 0% to 25% degradation for different applications, which indicates that these



(a) Memory stall time.

(b) Network traffic (flit-crossings).

(c) Execution time.

Figure 4.1 Comparison of MESI vs. DeNovo protocols without the communication optimizations. All bars are normalized to the corresponding ML protocol.

techniques are beneficial for some applications.

We divide the remainder of this section into two parts. In Section 4.5.1, we compare the performance of MESI and DeNovo without the communication optimizations. We also evaluate the effect of cache lines in these two protocols. In Section 4.5.2, we analyze the performance of the two communication optimizations.

4.5.1 Evaluation of Word and Line Protocols

Figures 4.1a, 4.1b, and 4.1c respectively show the memory stall time, network traffic, and the overall execution time for all the word and line protocols described in Section 4.2 for each application. Each bar (protocol) is normalized to the corresponding (state-of-the-art) MESI-line (ML) bar.

The memory stall time bars (Figure 4.1a) are divided into four components. The bottommost indicates time spent by a memory instruction stalled due to a blocked L1 cache related resource (e.g., the 64 entry buffer for non-blocking stores is full). The upper three indicate additional time spent stalled on an L1 miss that gets resolved at the L2, a remote L1 cache, or main memory respectively. The network traffic bars (Figure 4.1b) show the number of flit crossings through on-chip network routers due to reads, writes, writebacks, and invalidations respectively. The execution time bars (Figure 4.1c) are divided into time spent in compute cycles, memory stalls, and synchronization cycles respectively. We primarily focus on the memory stall time and the network traffic results because the overall execution time results follow a similar trend as the memory stall time results.

MESI vs. DeNovo word protocols (MW vs. DW): MW and DW are not practical protocols because of their excessive tag overhead. A comparison is instructive, however, to understand the efficacy of selective self-invalidation, independent of line-based effects such as false sharing. In all cases, DW's performance is competitive with MW. For the cases where it is slightly worse (e.g., Barnes), the cause is higher remote L1 hits in DW than in MW. This is because in MW, the first reader forces the last writer to writeback to L2. Thus, subsequent readers get their data from L2 for MW but need to go to the remote L1 (via L2) for DW, slightly increasing the memory stall time for DW. However, in terms of network traffic, DW always significantly outperforms MW.

MESI vs. DeNovo line protocols (ML vs. DL): DL shows about the same or better memory stall times as ML. For LU and kdTree-false, DL shows 55.5% and 73.6% reduction in memory stall time over ML, respectively. Here, DL enjoys one major advantage over ML: DL incurs no false sharing due to its per-word coherence state. Both LU and kdTree-false contain some false sharing, as indicated by the significantly higher remote L1 hit component in the memory stall time graphs for ML. In terms of network traffic, DL outperforms ML except for fluidanimate and radix. Here, DL incurs more network traffic because registration (write-traffic) is still at word-granularity (shown in 4.1b).

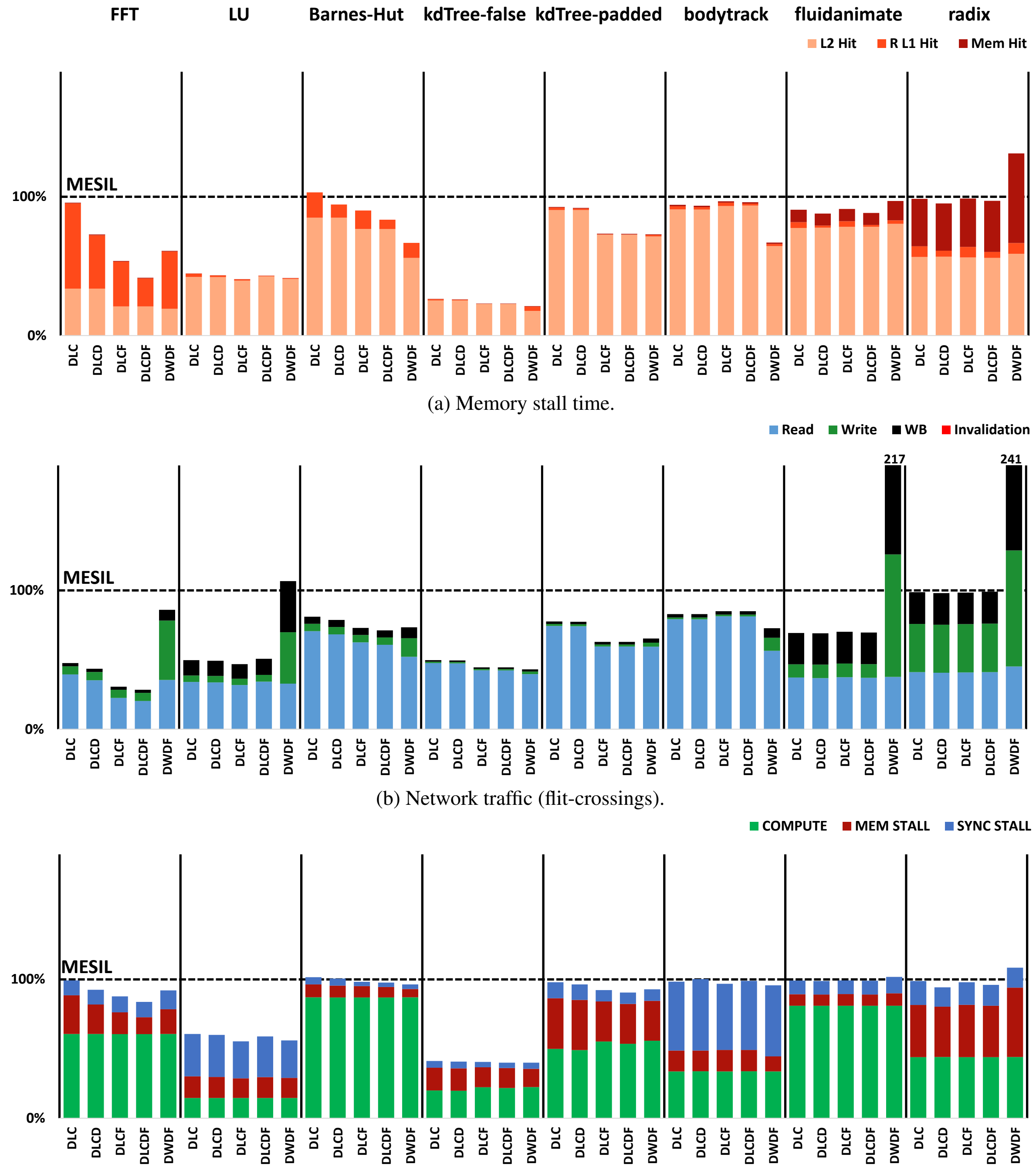
Effectiveness of write-combining optimization (DL vs. DLC): The additional registration traffic in DL can be mitigated with a "write-combining" optimization that aggregates individual registration requests at a cache-line granularity as described in Section 2.5. Compared to DL, DLC has minimal reduction in the memory stall time, but has significant impact on the store traffic. Specifically, the traffic anomaly for fluidanimate and radix in DL is completely eliminated and DLC outperforms ML for every application in network traffic.

Effectiveness of cache lines for MESI: Comparing MW and ML, we see that the memory stall time reduction resulting from transferring a contiguous cache line instead of just a word is highly application dependent. The reduction is largest for radix (a large 93%), which has dense arrays and no false sharing. Most interestingly, for kdTree-false (object-oriented AoS style with false sharing), the word based MESI does better than the line based MESI by 35%. This is due to the combination of false sharing and less than perfect spatial locality. Bodytrack is similar in that it exhibits little spatial locality due to its irregular access pattern. Consequently, ML shows higher miss counts and memory stall times than MW (due to cache pollution from the useless words in a cache line).

Effectiveness of cache lines for DeNovo: Comparing DW with DL, we see again the strong application dependence of the effectiveness of cache lines. However, because false sharing is not an issue with DeNovo, both LU and kdTree-false enjoy larger benefits from cache lines than in the case of MESI (77% and 62% reduction in memory stalls). Analogous to MESI, Bodytrack sees larger memory stalls with DL than with DW because of little spatial locality.

4.5.2 Evaluation of the Communication Optimizations

Figures 4.2a, 4.2b, and 4.2c respectively show the memory stall time, network traffic, and the overall execution time for all the protocols with either or both of the communication optimizations. We also show a bar for DLC as these optimizations are applied on top of it. The bars in these figures are also normalized to the corresponding MESI-line (ML) bar to appreciate how much overall benefit we get compared to the



(a) Memory stall time.

(b) Network traffic (flit-crossings).

(c) Execution time.

Figure 4.2 Performance evaluation of the communication optimizations. All bars are normalized to the corresponding ML protocol.

baseline ML protocol.

Effectiveness of direct cache-to-cache transfer with DLC: FFT and barnes exhibit much opportunity for direct cache-to-cache transfer. For these applications, DLCD is able to significantly reduce the remote L1 hit latencies when compared to DLC.

Effectiveness of flexible communication granularity with DLC: DLCF performs about as well or better than ML and DLC for all cases, except for Bodytrack. Bodytrack does not do as well because of the line granularity for cache allocation (addresses). DLCF can bring in data from multiple cache lines; although this data is likely to be useful, it can potentially replace a lot of allocated data. As we see later, flexible communication at word address granularity does much better for Bodytrack. Overall, DLCF shows up to 77% reduction in memory stall time over ML and up to 43% over DLC.

Effectiveness of combined optimizations with DLC: DLCDF combines the benefits of both DLCD and DLCF to show either about the same or better performance than all the other line based protocols.

Effectiveness of combined optimizations with DW: For applications like Bodytrack with low spatial locality, word-based protocols have the advantage over line based protocols by not bringing in potentially useless data and/or not replacing potentially useless data. We find that DW with our two optimizations (DWDF) does indeed perform better than DLCDF for this application. In fact, DDFW does better for 5 out of the 8 applications.

4.6 Summary

We provide performance analysis of the DeNovo protocol and its two optimizations that address inefficiencies related to coherence and communication. We ran our simulations on a 64-core system and compared the performance against the MESI protocol using seven benchmarks. We show that DeNovo including the two communication optimizations reduces the memory stall time by 32% on average. It also reduces the overall network traffic by 36% which results in direct savings in energy. These reductions in the memory stall time and the network traffic result in an overall reduction in execution time by 17% on average. Hence we show that by exploiting the features of a disciplined parallel programming model we not only simplify the coherence protocol but also provide better performance and energy compared to traditional hardware protocols.

CHAPTER 5

DATA STORAGE

In this chapter we focus on inefficiencies related to data storage. Scratchpads and hardware-managed caches are two widely used memory organizations in today's memory hierarchies especially in heterogeneous SoC systems. Caches are easy to program because they are largely transparent to the programmer. However, every cache access results in a TLB access and tag comparisons making it a power-inefficient organizational structure. Additionally, in modern object-oriented programming styles, where objects are generally composed of multiple fields, it is common for only a few of the object's fields to be accessed together. This problem in the context of network traffic wastage has been addressed by one of our proposed communication optimizations, flexible transfer granularity, in Chapter 2. As caches store data at fixed cache line granularities, the unused fields of the objects waste valuable cache space too.

Software-managed directly addressable scratchpad memories are commonly used in SoC designs because they are more energy and delay efficient compared to caches and are not susceptible to pathological performance anomalies due to conflict misses (especially for realtime systems) [14, 132, 133]. Scratchpads offer significant area and energy savings (e.g. 34% area and 40% power [16] or more [87]) because they do not require tag lookups or have misses like traditional caches: once the data is inserted into the scratchpad, it remains there until explicitly removed. This requires explicit addressing and programmer support, which can be useful because programmers often understand their data usage and layout, allowing them to compactly store only the necessary data.

However, scratchpads suffer from multiple inefficiencies. Since they comprise a disjoint address space from global memory, they use additional instructions to explicitly move data between the scratchpad and global memory. These instructions typically use the core's pipeline resources, use registers to bring data in and out of the scratchpad from/to the main memory, and pollute the cache. Data in the scratchpad is also visible only to the core it belongs to. As a result, the data in the scratchpad requires explicit, eager writebacks to the main memory to make the data globally visible. These explicit data movements and utilization of core resources result in unnecessary energy wastage and performance penalty.

To address the inefficiencies of caches and scratchpads we introduce a new memory organization called ***stash***, which combines the best properties of the cache and scratchpad organizations. Similar to a scratchpad, the stash is software managed and directly addressable. However, the stash also has a mapping between the global and local stash address spaces. This mapping is provided by the software and is used by the hardware whenever a translation between the two address spaces is required (e.g., miss, writeback, and remote requests). The mapping allows the stash to avoid the explicit data movement of the scratchpad and instead implicitly move data between address spaces. Furthermore, like a scratchpad the stash can compactly map non-contiguous global memory elements and obtain the benefits of AoS to SoA transformation without any software changes or actual data transformations in memory. Similar to a cache, the stash values are globally visible. Global visibility requires extensions to the coherence protocol. The functionality of the stash organization is not dependent on a particular coherence protocol. Different coherence protocols can be extended with different tradeoffs to support stash. In Section 5.4.4, we describe these tradeoffs for MESI and DeNovo. In our implementation we chose DeNovo for coherence support for its simplicity and low overhead.

There has been a significant amount of prior work on optimizing the behavior of private memories. This includes methods for transferring data from the memory to the scratchpad directly without polluting registers or caches [10, 19, 70], changing the data layout for increased compaction [36, 42], removing tag checks for caches [120, 145], and virtualizing private memories [48, 49, 50, 91]. While each of these techniques mitigates some of the inefficiencies of scratchpads or caches, none of these techniques mitigates all of the inefficiencies. While a quantitative comparison with all of the techniques is outside the scope of this thesis, we provide a detailed qualitative comparison for all (Section 7.3.1) and quantitatively compare our results to the closest technique to our work: scratchpad enhanced with a DMA engine.

Feature	Benefit	Cache	Scratchpad	Stash
Directly addressed	No address translation hardware access	✗ (if physically tagged)	✓	✓ (on hits)
	No tag access	✗	✓	✓
	No conflict misses	✗	✓	✓
Compact storage	Efficient use of SRAM storage	✗	✓	✓
Global addressing	Implicit data movement from/to structure No pollution of other memories On-demand loads into structures	✓	✗	✓
Global visibility	Lazy writebacks Reuse across kernels and phases	✓	✗	✓

Table 5.1 Comparison of cache, scratchpad, and stash.

5.1 Background

5.1.1 Caches

Caches are a common memory organization in modern systems. Their transparency to software makes them easy to program, but incurs some inefficiencies.

Indirect, hardware-managed addressing: Cache loads and stores specify addresses that hardware must translate to determine the physical location of the accessed data. This *indirect addressing* implies that each cache access (a hit or a miss) incurs (energy) overhead for TLB lookups and tag comparisons. Virtually tagged caches do not require TLB lookups on hits, but they incur additional overhead, including dealing with synonyms, page mapping and protection changes, and cache coherence [17]. Further, the indirect, hardware-managed addressing also results in unpredictable hit rates due to cache conflicts, causing pathological performance (and energy) anomalies, a particularly notorious problem for real-time systems.

Inefficient, cache line based storage: Caches store data at fixed cache line granularities which wastes SRAM space when a program does not access the entire cache line (e.g., when a program phase traverses an array of large objects but accesses only one field in each object).

5.1.2 Scratchpads

Scratchpads are local memories that are managed in software, either by the programmer or through compiler support. Unlike caches, scratchpads are directly addressed, without the energy overheads of TLB lookups and tag comparisons. Direct, software-managed addressing also eliminates the pathologies of conflict misses, providing a predictable (100%) hit rate. Finally, scratchpads allow for a compact storage layout as the software only brings useful data into the scratchpad. Scratchpads, however, suffer from other inefficiencies.

Not Globally Addressable: Scratchpads have a separate address space disjoint from the global address space, with no mapping between the two. To exploit the benefits of the scratchpad for globally addressed data, extra instructions must be used to *explicitly* move such data between the two spaces, incurring performance and energy overhead. Furthermore, in current systems the additional loads and stores typically move data via the core's L1 cache and its registers, *polluting* these resources and potentially evicting (spilling) useful data. Scratchpads also do not perform well for applications with *on-demand loads* because today's scratchpads usually pre-load all elements before they are accessed. In applications with control/data dependent accesses, only a few of those pre-loaded elements will be accessed.

Not Globally Visible: A scratchpad is visible only to its local core. Therefore dirty data must be explicitly written back to a global address (and flushed) before it is needed by other cores in subsequent kernels. [1] In current GPUs, such writebacks typically occur before the end of the kernel (when the scratchpad space is deallocated), even if the same data may be reused in a later phase of the program. (GPU codes assume data-race-freedom; i.e., global data moved to/from the scratchpad is not concurrently written/read in the same kernel.) Thus, the lack of global visibility results in potentially unnecessary, *eager writebacks* and *precludes reuse* of data across multiple kernels.

Table 5.1 compares caches and scratchpads (Section 5.2 discusses the stash column).

Example and usage modes: Figure 5.1a shows an example to demonstrate the above inefficiencies. The code at the top reads one field, $fieldX$ (of potentially many), from an array of structs (AoS) data structure, $aosA$, performs some computation using this field, and writes back the result to $aosA$. The bottom of the figure shows some of the corresponding steps in hardware. First, the program must *explicitly load* a copy of the data into the scratchpad from the corresponding global address (event 1; additionally events 2 and 3 on an L1 miss). This explicit load will bring the data into the L1 cache (hence *polluting* it as a result). Next, the data must be brought from the L1 cache into a local register in the core (event 4) so the value can be explicitly stored into the corresponding scratchpad address. At this point, the scratchpad is populated with the global value and the program can finally use the data in the scratchpad (events 6 and 7). Once the program is done modifying the data, the dirty scratchpad data is *explicitly written back* to the global address space, requiring loads from the scratchpad and stores into the cache (not shown in the figure).

[1] A kernel is the granularity at which the CPU invokes the GPU and it executes to completion on the GPU.

```
func_scratch(struct* aosA, int myOffset, int myLen)
{
  __scratch__ int local[myLen];
  // explicit global load and scratchpad store
  parallel for(int i = 0; i < myLen; i++) {
    local[i] = aosA[myOffset + i].fieldX;
  }
  // do computation(s) with local(s)
  parallel for(int i = 0; i < myLen; i++) {
    local[i] = compute(local[i]);
  }
  // explicit scratchpad load and global store
  parallel for(int i = 0; i < myLen; i++) {
    aosA[myOffset + i].fieldX = local[i];
  }
}
```

```
func_stash(struct* aosA, int myOffset, int myLen)
{
  __stash__ int local[myLen];
  //AddMap(stashBase, globalBase, fieldSize,
           objectSize, rowSize, strideSize,
           numStrides, isCoherent)
    AddMap(local[0], aosA[myOffset], sizeof(int),
         sizeof(struct), myLen, 0, 1, true);


  // do computation(s) with local(s)
  parallel for(int i = 0; i < myLen; i++) {
    local[i] = compute(local[i]);

  }
}
```

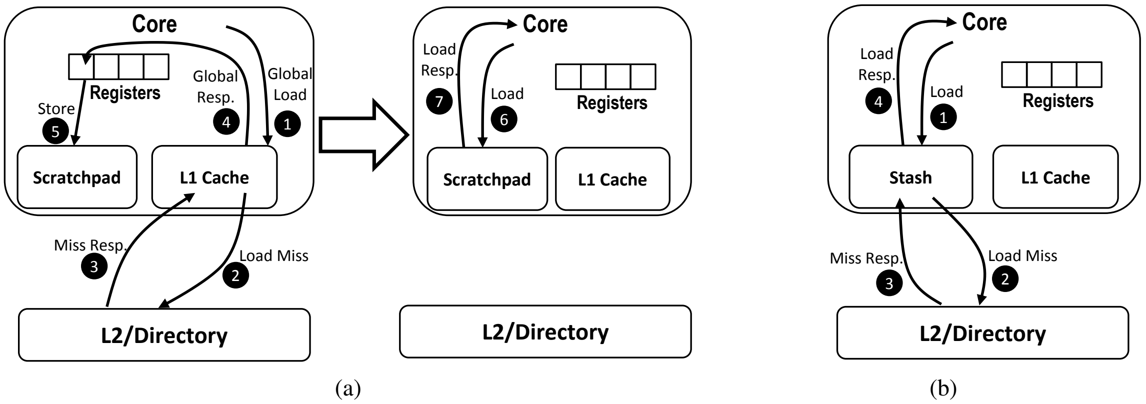


Figure 5.1 Codes and hardware events for (a) scratchpad and (b) stash. Scratchpads require instructions for explicit data movement to/from the global space. Stash uses $AddMap$ to provide a mapping to the global space, enabling implicit movement.

We call the above usage mode where data is moved explicitly from/to the global space as *global-unmapped* mode. Scratchpads can also be used for private, temporary values. Such values do not require global address loads or writebacks as they are discarded after their use (they trigger only events 6 and 7 in the figure). We call this mode as *temporary* mode.

5.2 Stash Overview

A stash is a new SRAM organization that combines the advantages of scratchpads and caches. Table 5.1 summarizes the benefits of the stash, showing that it combines the best of both caches and scratchpads. It has the following features.

Directly addressable: Like scratchpads, a stash is directly addressable and data in the stash is explicitly allocated by software (either the programmer or the compiler).

Compact storage: Since it is software managed, only data that software deems useful is brought into the stash. Thus, like scratchpads, stash enjoys the benefit of a compact storage layout, and unlike caches, it is not susceptible to storing useless words of a cache line.

Physical to global address mapping: In addition to being able to generate a direct, physical stash address, software can also specify a mapping from a contiguous set of stash addresses to a (possibly non-contiguous) set of global addresses. Our architecture can map to a 1D or 2D, possibly strided, tile of global addresses.[2] Hardware maintains the mapping from the stash to global space until the data is present in the stash.[3]

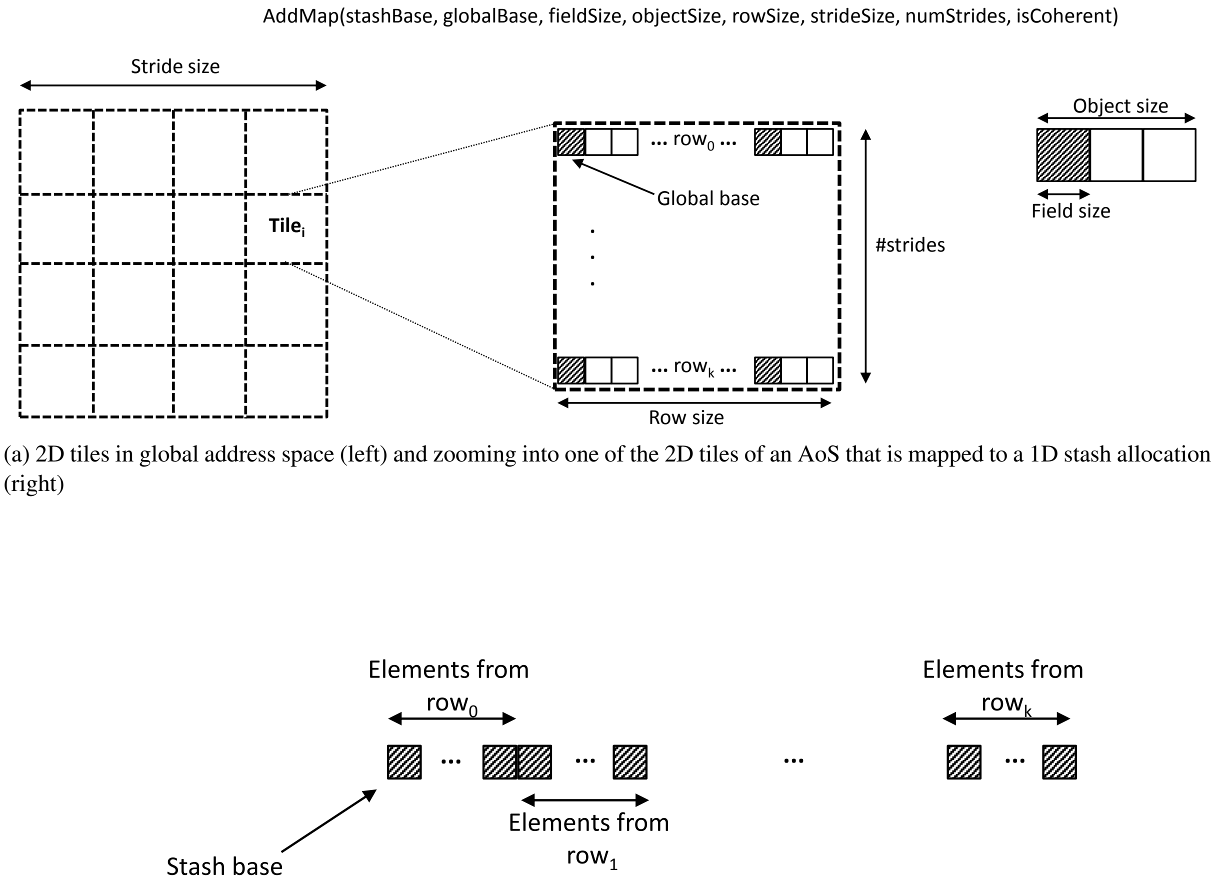
Global visibility: Like a cache, stash data is globally visible through a coherence mechanism (described in Section 5.4.4). A stash, therefore, does not need to eagerly writeback dirty data. Instead, data can be reused and lazily written back only when software actually needs the stash space to allocate new data (similar to cache replacements). If another core needs that data, it will be forwarded to the stash through the coherence mechanism. In contrast, for scratchpads in current GPUs, data is written back to global memory (and flushed) at the end of a kernel, resulting in potentially unnecessary and bursty writebacks with no reuse across kernels.

The first time a load occurs to a newly mapped stash address, it implicitly moves the data from the mapped global space to the stash and returns it to the core (analogous to a cache miss). Subsequent loads for that address immediately return the data from the stash (analogous to a cache hit, but with the energy benefits of direct addressing). Similarly, no explicit stores are needed to write back the stash data to its mapped global location. Thus, the stash enjoys all the benefits of direct addressing of a scratchpad on its hits (which occur on all but the first access), but without the overhead incurred by the additional loads and stores required for explicit data movement in the scratchpad.

Figure 5.1b shows the code from Figure 5.1a modified for a stash. The stash code does not have any explicit instructions for moving data into or out of the stash from/to the global address space. Instead, the stash has an $AddMap$ call that specifies the mapping between the two address spaces (further discussed in Section 5.3). In hardware (bottom part of the figure), the first load to a stash location (event 1) implicitly triggers a global load (event 2) if the data is not already present in the stash. Once the load has returned the desired data (event 3), it is sent to the core (event 4). Subsequent accesses will directly return the data from the stash without consulting the global mapping.

[2] Our design can be easily extended to other access patterns with additional parameters and is not fundamentally restricted.

[3] This is the reverse of today's virtual to physical translation.



(a) 2D tiles in global address space (left) and zooming into one of the 2D tiles of an AoS that is mapped to a 1D stash allocation (right)

(b) 1D stash space with one of the fields of the AoS

Figure 5.2 Mapping a global 2D AoS tile to a 1D stash address space

5.3 Stash Software Interface

We envision the programmer or the compiler will map a part of the global address space to the stash. Programmers writing applications for today's GPU scratchpads already effectively compute such a mapping. There has also been prior work on compiler methods to automatically deduce this information [14, 77, 104]. The mapping of the global address space to stash requires strictly less work compared to that of a scratchpad as it avoids the need for explicit loads and stores between the global and stash address spaces. Reducing the programmer overhead or compiler analysis to automatically generate this mapping is outside the scope of this paper. Instead, we focus on the hardware-software interface for stash and we use GPU applications that already contain scratchpad related annotations in our experiments (Section 6).

5.3.1 Specifying Stash-to-Global Mapping

The mapping between global and stash address spaces is specified using two intrinsic functions. The first intrinsic, $AddMap$, is called when communicating a new mapping to the hardware. We need an $AddMap$ call for every data structure (a linear array or a 2D tile of an AoS structure) that is mapped to the stash per thread block.

Figure 5.1b shows an example usage of $AddMap$ along with its definition. To better understand the parameters of $AddMap$ we show how an example 2D global address space is mapped to a 1D stash address space in Figure 5.2. First, the global address space is divided into multiple tiles as shown in Figure 5.2a. Each of these tiles has a virtual address base that is mapped to a corresponding stash address base at runtime. The first two parameters of $AddMap$ specify stash and global virtual base addresses of a given tile. The stash base address in the $AddMap$ call is local to the thread block. This local stash base address is mapped to the actual physical address at runtime. This mapping is similar to how today's scratchpads are scheduled. Figure 5.2a also shows the various parameters used to describe the object and the tile. The field size and the object size provide information about the global data structure (field size = object size for scalar arrays). The next three parameters specify information about the tile in the global address space: the row size of the tile, global stride between the two rows of the tile, and the number of strides. Finally, Figure 5.2b shows the 1D mapping of the individual fields of interest from the 2D global AoS data structure. The last field of $AddMap$, $isCoherent$, indicates the operation mode of the stash, discussed in Section 5.4.4.

The second intrinsic, $ChgMap$, is used whenever there is a change in mapping or the operation mode of the chunk of global addresses that are mapped to the stash. The parameters for the $ChgMap$ call are the same as for the $AddMap$ call.

5.3.2 Stash Load and Store Instructions

The load and store instructions for a stash access are similar to those for a scratchpad. On a hit, the stash needs to just know the requested address. On a miss, in addition to the requested address, the stash needs to know which stash-to-global mapping (an index in a hardware table, discussed later) it needs to apply. This information can be encoded in the instruction in at least two different ways without requiring extensions to current ISA. CUDA, for example, has multiple address modes for LD/ST instructions - register, register-plus-offset, and immediate addressing. The register based addressing schemes hold the stash (or scratchpad) address in the $register$ field. We can use the higher bits of register for storing the map index (since a stash address does not need all the bits of the register). Alternatively, we can use the register-plus-offset addressing scheme, where register holds the stash address and $offset$ holds the map index (in CUDA, offset is currently ignored when the local memory is configured as a scratchpad). Section 5.4 discusses more details regarding the hardware map index and how it is used.

5.3.3 Usage Modes

A single mapping for the stash data usually holds the translation for a large, program-defined chunk of data; each such chunk can be used in four different modes:

Mapped Coherent: This mode is based on the description so far – it provides a stash-to-global mapping and the stash data is globally visible and coherent.

Mapped Non-coherent: This mode is similar to that of "Mapped Coherent" except that the stash data is not coherent. The stash can still avoid having explicit instructions to move the data from the global address space to the stash, but any modifications to local stash data are not reflected to the global address space.

Global-unmapped and Temporary: These two modes are similar to that of scratchpad as described in Section 5.1.2. If for some reason, a given chunk of global address space cannot be mapped to stash using the $AddMap$ call, the program can always fall back to the way scratchpads are currently used. This enables using all current scratchpad code in our system.

5.3.4 Stash Allocation Management

Two thread blocks, within a given process or across processes, get different stash allocations when they are scheduled. So two stash addresses from two different thread blocks never point to the same location in the stash. Each thread block sub divides its stash allocation across multiple data structures that it accesses in a given kernel. These sub divisions can be in any one of the usage modes described earlier. As a result, a given stash location corresponds to only a single mapping, if any. So a given stash address cannot map to two global addresses at the same time. But a given global address can be replicated at multiple stash addresses (as long as the data-race freedom assumption holds). This is handled by the coherence mechanism (Section 5.4.4).

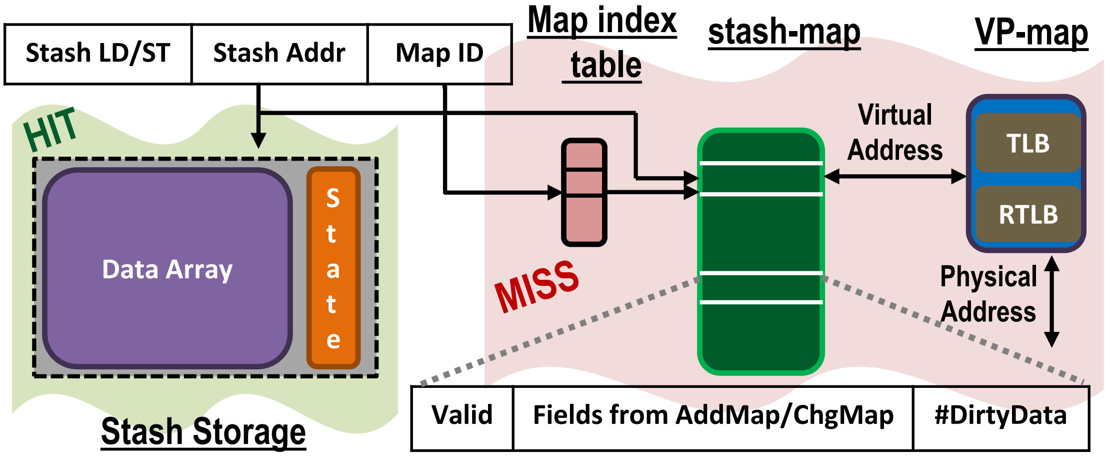


Figure 5.3 Stash hardware components.

5.4 Stash Hardware Design

This section describes the design of the stash hardware, which is aimed primarily at providing stash-to-global address translations and vice versa for misses, writebacks, and remote requests. The next three sub-sections describe the stash hardware components, the stash hardware operations, and hardware support for stash coherence respectively.

5.4.1 Stash Components

The stash consists of four hardware components shown in Figure 5.3: (1) *stash storage*, (2) *stash-map*, (3) *VP-map*, and (4) *map index table*. This section briefly describes each component; the next section describes how they together enable the different stash operations in more detail.

Stash Storage

This component consists of data storage similar to a scratchpad. It also has storage for per-word state to identify hits and misses (dependent on the coherence protocol) and state bits for writebacks.

Stash-Map

The stash-map contains an entry for each mapped stash data partition. An entry contains the information to translate between the stash and the global virtual address space (as specified by an $AddMap$ of $ChgMap$

call). For GPUs, there is one such entry per thread block (or workgroup). Stash, similar to a scratchpad, is partitioned among multiple thread blocks. The scheduling of these thread blocks on a given core (and allocation of specific stash space) occurs only at runtime. Hence, each stash-map entry needs to capture thread block specific mapping information including the runtime stash base address provided by $AddMap$ (or $ChgMap$). Figure 5.3 shows a stash-map entry with fields from $AddMap$/$ChgMap$ and two additional fields: a $Valid$ bit and a $\#DirtyData$ field used for writebacks. We can pre-compute much of the information required for the translations and do not need to store all the fields in the hardware (see Section 5.4.3).

The stash-map can be implemented as a circular buffer with a tail pointer. Our design makes sure that the entries in stash-map are added and removed in the same order for easy management of stash-map's fixed capacity. The number of entries should be at least the maximum number of thread blocks a core can execute in parallel times the maximum number of $AddMap$ calls allowed per thread block. We found that applications did not use more than four map entries simultaneously. So assuming eight thread blocks used in parallel, 64 map entries were sufficient.

VP-map

A stash-to-global mapping can span multiple virtual pages. We need virtual-to-physical translations for each such page to move data (implicitly) between stash and main memory. VP-map uses two structures for this purpose. The first structure, TLB, provides a virtual to physical translation for every mapped page, required for stash misses and writebacks. We can leverage the core's TLB for this purpose. For remote requests which come with a physical address, we need a reverse translation from the physical page number to the virtual page number. The second structure, $RTLB$, provides this reverse translation and is implemented as a CAM over physical page numbers. The TLB and RTLB can be merged into a single structure, if needed, to reduce area.

Each entry in the VP-map has a pointer (not shown in Figure 5.3) to an entry in the stash-map that indicates the latest stash-map entry that requires the given translation. When a stash-map entry is replaced, any entries in the VP-map that have a pointer to the same map entry are also removed as this translation is no longer needed. By keeping the $RTLB$ entry (and the TLB entry if kept separate from system TLB) around until the last mapping that uses it is removed, we guarantee that we never miss in the $RTLB$.

Map Index Table

The map index table per thread block gives an index into the thread block's stash-map entries. An $AddMap$ allocates an entry into the map index table. Assuming a fixed ordering of $AddMap$ calls, the compiler can determine which table entry corresponds to a mapping – it includes this entry's ID in future stash instructions corresponding to this mapping (using the format in Section 5.3). The size of the table is the maximum number of $AddMap$s allowed per thread block (our design allocates four entries per thread block). If the compiler runs out of these entries, it cannot map any more data to the stash.

5.4.2 Operations

We next describe in detail how different stash operations are implemented.

Hit: On a hit (determined by coherence bits as discussed in Section 5.4.4), the stash acts like a scratchpad, accessing only the storage component.

Miss: A miss needs to translate the stash address into a global physical address. It uses the map table index provided by its instruction to determine its stash-map entry. Given the stash address and the stash base from the stash-map entry, we can calculate the stash offset. Using the stash offset and the other fields of the stash-map entry, we can calculate the virtual offset (details in Section 5.4.3). Once we have the virtual offset, we can add it to the virtual base of the stash-map entry providing us with the corresponding global virtual address for the given stash address. Finally, using the VP-map we can determine the corresponding physical address which is used to handle the miss.

Additionally, a miss must consider if the data it replaces needs to be written back and a store miss must perform some bookkeeping to facilitate a future writeback. These actions are described next.

Lazy writebacks: On a store, we need to maintain the index of the current stash-map entry for a future lazy writeback. A simple implementation will store the stash-map entry's index per word which is not space efficient. Instead, we store the index at the granularity of a larger chunk of stash space, say 64B, and perform writebacks at the same granularity.[4] To know when to update this per chunk stash-map index, we have a dirty bit per stash chunk. On a store miss, if this dirty bit is not set, we set it and update the stash-map index. In addition to updating the stash-map index, we also update the $\#DirtyData$ counter of the stash-map entry to track the number of dirty stash chunks in the corresponding stash space. The per chunk dirty bits

[4] One of the side effects of this approach is that the data structures need to be aligned at the chosen chunk granularity.

are unset at the end of the thread block.

Lazy writebacks require recording that a stash word needs to be lazily written back. We use an additional bit per chunk to indicate the need to writeback. This bit is set for all the dirty stash chunks at the end of a thread block. Whenever a new mapping needs a stash location that was marked to be written back, we use the per chunk stash-map index to access the stash-map entry – similar to a miss, this allows us to determine the physical address to which the writeback is sent. A writeback of a word in a chunk triggers a writeback of all the dirty words (leverage coherence state to determine which words are dirty) in the chunk. On a writeback, the $\#DirtyData$ counter of the map entry is decremented. When the counter reaches zero, the map entry is marked as invalid.

AddMap: An $AddMap$ call advances the stash-map's tail, and sets the next entry of its thread block's map index table to point to this tail entry. It updates the stash-map tail entry with its parameters and sets it to $Valid$ (Section 5.4.1). It also deletes any entries from the VP-map that has the new stash-map tail as the back pointer.

If the new stash-map entry was previously valid, then it indicates an old mapping that is no longer being used, but still has dirty data that has not yet been (lazily) written back. We initiate these writebacks and block the core until they are done. Alternately, a scout pointer can stay a few entries ahead of the tail, triggering non-blocking writebacks for valid stash-map entries. This case is rare because we expect a new mapping to have already reclaimed the stash space held by the old mapping, writing back the old dirty data on replacement. The above process ensures that the entries in stash-map are removed in the same order that they were added so that we can guarantee we never miss in RTLB for remote requests.

Finally, for every virtual page mapped, an entry is added to the $RTLB$ (and TLB if maintained separately in addition to system's TLB) of VP-map. If the system TLB has the physical translation for this page, we populate the corresponding entries in VP-map. If the translation does not exist in the TLB, the physical translation is acquired at the subsequent stash miss. For every virtual page in the current map, the stash-map pointer in the corresponding entries in VP-map is updated to point to the current map entry. In the unlikely scenario where the VP-map becomes full and has no more space for new entries, we advance the tail of the stash-map (along with performing any necessary writebacks) until at least one entry in VP-map is removed.

ChgMap: $ChgMap$ updates a current stash-map entry with new mapping information for given stash data. If isCoherent is modified from true to false, then we need to issue writebacks for the old mapping. Instead, if it is modified from false to true, we need to issue ownership/registration requests for all dirty words in the old mapping according to the coherence protocol employed (Section 5.4.4).

5.4.3 Address Translation

```
/* Values that can be precomputed */
//stashBytesPerRow = (rowSize / objectSize) * fieldSize
//virtualToStashRatio = strideSize / stashBytesPerRow
//objectToFieldRatio = objectSize / fieldSize

/* stashBase is obtained from the stash-map entry */
stashOffset = stashAddress - stashBase

fullRows = stashOffset * virtualToStashRatio
lastRow = (stashOffset % stashBytesPerRow) * objectToFieldRatio
virtualOffset = fullRows + lastRow

/* virtualBase is obtained from the stash-map entry */
virtualAddress = virtualBase + virtualOffset
```

Listing 5.1 Translating stash address to virtual address

Listing 5.1 shows the logic for translating a stash offset to its corresponding global virtual offset. The stash offset is obtained by subtracting a stash address (provided with the instruction) from the stash base found in the stash-map entry. As shown in the translation logic, we do not need to explicitly store all the parameters of an $AddMap$ call in the hardware. We can pre-compute information required for the translations. For example, for stash to virtual translation, we can pre compute three values. First, we need the number of bytes in stash that a given row in the global space corresponds to. This is equal to the number of bytes the shaded fields in a given row amount to in Figure 5.2a. This value is stored in $stashBytesPerRow$. Next to account for the gap between the two global rows, we need to know the global span of $stashBytesPerRow$. So we calculate the ratio of $strideSize$ to $stashBytesPerRow$. This value is stored in $virtualToStashRatio$. $virtualToStashRatio$ helps us find the corresponding global row for a given stash address. Finally, to calculate the relative position of the global address within the identified row, we need the ratio of object size to field size, stored in $objectToFieldRatio$. Using these pre-computed values, we can get the corresponding virtual span (virtual offset) for all the full rows of the tile the stash offset spans and for the partial last row (if any). This virtual offset is added to the virtual base found in the stash-map entry to get the corresponding virtual address. Overall, we need six arithmetic operations per miss.

```
/* Values that can be precomputed */
//stashBytesPerRow = (rowSize / objectSize) * fieldSize
//stashToVirtualRatio = stashBytesPerRow / strideSize
//fieldToObjectRatio = fieldSize / objectSize

/* virtualBase is obtained from the stash-map entry */
virtualOffset = virtualAddress - virtualBase

fullRows = virtualOffset * stashToVirtualRatio
lastRow = (virtualOffset % rowSize) * fieldToObjectRatio
stashOffset = fullRows + lastRow

/* stashBase is obtained from the stash-map entry */
stashAddress = stashBase + stashOffset
```

Listing 5.2 Translating virtual address to stash address

The logic for the reverse translation of global address to stash address is similar and is shown in Listing 5.2.

5.4.4 Coherence Protocol Extensions for Stash

All *Mapped Coherent* stash data must be kept coherent. We can use any coherence protocol or extend it. We can either use a traditional hardware protocol such as MESI[5] or a software-driven hardware coherence protocol like DeNovo (Chapter 2), as long as it supports the following three features:

[5] We assume data-race-free programs.

Tracking at word granularity: Stash data must be tracked at word granularity because only useful words from a given cache line are brought into the stash.[6]

Merging partial cache lines: When the stash sends data to a cache (either as a writeback or a remote miss response), it may send only part of a cache line. The the cache must be able to merge partial cache lines.

Map index for physical-to-stash mapping: When data is modified by a stash, the directory needs to record the modifier core (as usual) and also the stash-map entry for that data (so that a remote request to that data can easily determine where to obtain it from the stash).

We can support the above features in a traditional single-writer directory protocol (e.g., MESI) with minimal overhead by retaining coherence state at line granularity, but adding a bit per word to indicate whether its up-to-date copy is present in a cache or a stash. Assuming a shared last level cache (LLC), when a directory receives a stash store miss request, it transitions to modified state for that line, sets the above bit in the requested word, and stores the stash-map index (obtained with the miss request) in the data field for the word at the LLC. Although this is a straightforward extension, it is susceptible to false-sharing (and the stash may lose the predictability of a guaranteed hit after an initial load). To avoid false sharing, we could use a sector-based cache with word sized sectors, but this incurs heavy overhead (state bits and sharers list per word at the directory).

Sectored protocols: Alternatively, we can use DeNovo from Chapter 2 that already has word granularity sectors (coherence state is at word granularity, but tags are at conventional line granularity). Since such sectored protocols already track coherence state per word, they do not need the above extra bit to indicate whether the word is in a cache or stash – in modified state, the data field of the word in the LLC can encode the core where the data is modified, whether it is in the stash or cache at the core, and the stash-map entry in the former case.

Table 5.4.4 summarizes the storage overhead discussion above to support stash for variants of the MESI protocol (including the word based protocol) and the DeNovo protocol.

For this work, without loss of generality, we extended the DeNovo protocol for its simplicity. We extended the line based DeNovo protocol from Chapter 2 (with line granularity tags and word granularity coherence), originally proposed for multicore CPUs and deterministic applications, to work with heterogeneous CPU-GPU systems with stashes at the GPUs. We do not use the *touched* bit and regions in our

[6] We can support byte granularity accesses as long as all (stash-allocated) bytes in a word are accessed by the same core at a time; i.e., there are no word level data races. The benchmarks we have studied do not have byte granularity accesses.

Protocol	Tag Overhead	State Overhead	Sharer's List Overhead	Stash Map Overhead
MESI word	16 * N	16 * 5 = 80	16 * P	0
MESI line	N	5	P	16
MESI line with 16 sectors	N	16 * 5 = 80	16 * P	0
DeNovo line	N	2 + 16 = 18 (Section 2.7)	0	0

Table 5.2 Storage overheads (in bits) at the directory to support stash for various protocols. These calculations assume 64 byte cache lines with 4 byte words, N = tag bits, P = number of processors, and five state bits for MESI. We assume that the map information at the L2 can reuse the L2 data array (the first bit indicates if the data is a stash map index or not, the rest of the bits hold the index).

extensions. Although later versions of DeNovo support non-deterministic codes [130], our applications are deterministic. Further, although GPUs support non-determinism through operations such as atomics, these are typically resolved at the shared cache and are trivially coherent. Our protocol requires the following extensions to stash operations:

Stores: The DeNovo coherence protocol has three states, similar to that of the MSI protocol. Stores are considered a miss when in $Shared$ or $Invalid$ state. All store misses need to obtain registration (analogous to MESI's ownership) from the directory. In addition to registering the core ID at the directory, registration requests for words in the stash will also include the ID of the entry in the map.

Self-invalidations: At the end of a kernel we keep the data that is registered by the core (specified by the coherence state) but self-invalidate the rest of the entries to make the stash space ready for any future new allocations. In contrast, a scratchpad invalidates all the entries (after explicitly writing the data to the global address space).

Remote requests: Remote requests for stash that are redirected via the directory come with a physical address and a stash-map index (stored at the directory during the request for registration). Using the physical address, VP-map provides us with the corresponding virtual address. Using the stash-map index, we can obtain all the mapping information from the corresponding stash-map entry. We use the virtual base address from the entry and virtual address from the VP-map to calculate the virtual offset. Once we have the virtual offset and all other fields of the map entry, we can calculate the stash offset (translation logic in Listing 5.2), add it to the stash base, giving us the stash address.

5.4.5 Stash Optimization: Data Replication

It is possible for two allocations in stash space to be mapped to the same global address space. This can happen if the same read-only data is simultaneously mapped by several thread blocks in a core, or if data mapped in a previous kernel is mapped again in a later kernel on the same core. By detecting this replication and copying replicated data between stash mappings, it is possible to avoid costly requests to the directory.

To detect data replication, we can make the map a CAM, searchable on the virtual base address. On an $AddMap$ (an infrequent operation), the map is searched for the virtual base address of the entry being added to the map. If there is a match, we compare the tile specific parameters to confirm if the two mappings indeed perfectly match. If there is a match, we set a bit, $reuseBit$, and add a pointer to the old mapping in the new map entry. On a load miss, if the reuseBit is set, we first check the corresponding stash location of the old mapping and copy the value over if present. If not, we issue a miss to the directory.

If the new map entry is non-coherent and both the old and new map entries are for the same allocation in the stash, we need to writeback the old data. Instead, if the new map entry is coherent and both the old and new map entries are for different allocations in the stash, we need to send new registration requests for the new map entry.

5.5 Summary

Caches and scratchpads are two popular memory organization units. Caches are easy to program but are power-inefficient with TLB accesses, tag comparisons, and non-compact data storage. In contrast to caches, a scratchpad is a software managed, directly addressable memory that does not incur energy overheads of tag and TLB lookups, does not incur performance pathologies from conflict misses, and does not need to store data at the granularity of cache lines. However, scratchpads are only locally visible resulting in explicit movement of data between the global address space and the scratchpad, pollution of L1 caches, loss of performance with on-demand accesses, and no data reuse across kernels.

We proposed a new structure called the stash that is a hybrid of a cache and a scratchpad. Like a scratchpad, it is directly addressable and provides compact data storage. Like a cache, stash is a globally visible unit and has a mapping to global memory. Stash data can be copied implicitly without the overhead of additional instructions purely for data transfer. Stash also does not pollute the L1 cache and does not suffer from on-demand accesses. Stash facilitates lazy writebacks and thus, has data reuse across kernels. As a result the stash combines the benefits of both caches and scratchpads. One of the usage modes of stash, *Mapped Coherent*, requires that the data be kept coherent with the rest of the system. We describe three features a coherence protocol should support to be applicable for stash. In our implementation, we employed the DeNovo coherence protocol introduced in Chapter 2 with minor extensions (Section 5.4.4).

In the next chapter (Chapter 6) we evaluate the performance of stash compared to scratchpad and cache organizations for several microbenchmarks and applications. We also provide some future directions that can take the stash organization to other levels in the memory hierarchy and study storage inefficiencies of organizations beyond caches and scratchpads in Chapter 8.

CHAPTER 6

PERFORMANCE EVALUATION OF THE STASH ORGANIZATION

We extend the simulator infrastructure described in Section 4.1 for evaluating the stash organization. In the following sections, we describe these extensions to build a tightly coupled heterogeneous simulator. We also describe the performance and energy evaluations of the stash organization compared to the scratchpad and the cache organizations.

6.1 Simulator Infrastructure

We created an integrated CPU-GPU simulator using the system described in Section 4.1 to model the CPU and GPGPU-Sim v3.2.1 [15] to model the GPU. We use Garnet [8] to model a 4x4 mesh interconnect that has a GPU CU or a CPU core at each node. We use CUDA 3.1 [2] for the GPU kernels in the applications since this is the latest version of CUDA that is fully supported in GPGPU-Sim. Table 6.3 summarizes the key parameters of our simulated systems. We chose a slightly different configuration for the CPU compared to the one used for evaluating the DeNovo protocol (e.g., cache sizes and number of cores). This is largely dependent on the working set sizes of the applications we ran and being able to run them in reasonable simulation time. Our GPU is similar to an NVIDIA GTX 480.

As shown in Listing 5.1, we need six arithmetic operations for each translation between stash and global addresses. These operations need to be performed sequentially and the latency can be hidden by pipelining multiple requests for translation. So we do not model the latency of the stash hardware in our simulations but we do model its energy.

For energy comparisons, we extended GPUWattch [85] to measure the energy of the GPU CUs and

Hardware Unit	Hit Energy	Miss Energy
Scratchpad	5.53E-11J	–
Stash	5.54E-11J	8.68E-11J
L1 cache (8-way assoc.)	1.77E-10J	1.97E-10J
TLB access	1.41E-11J	1.41E-11J[2]
L1 cache + TLB	1.91E-10J	2.11E-10J

Table 6.1 Per access hit and miss energies (in Joules) for various hardware units.

the memory hierarchy including all stash components. To model stash storage, we extended the model for scratchpad available in GPUWattch to add additional bits for state information. We modeled the stash-map as an SRAM structure, VP-map as a CAM unit, and an ALU for each operation in the translation is modeled using an in-built ALU model in GPUWattch. The sizes for these hardware components are listed in Table 6.3. For our simulated system, GPUWattch estimated the peak dynamic power per GPU core to be 2.473W of which 0.13W is contributed by stash (0.09W by stash storage and the rest by its other hardware components).

We also use McPAT v1.1 [88] for our NoC energy measurements.[1] We do not measure the CPU core or the CPU L1 caches as our proposed stash design is implemented on the GPU. But we do charge the network traffic that originates from and destined to the CPU so that we measure any variations in the network caused by stash.

6.2 Baseline Heterogeneous Architecture

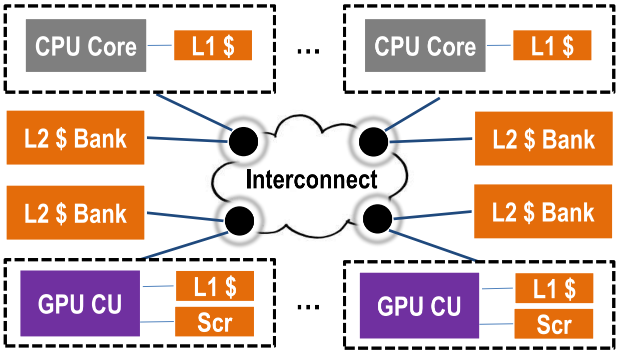


Figure 6.1 Baseline integrated architecture.

[1] We use McPAT's NoC model instead of GPUWattch's because our tightly coupled system more closely resembles a multicore system's NoCs.

[2] We do not model a TLB miss, so all our TLB accesses are charged as if they are hits.

Figure 6.1 shows our baseline heterogeneous architecture. It is a tightly integrated CPU-GPU system with a unified shared memory address space and coherent caches. The system is composed of multiple CPU and GPU cores, which are connected via an interconnection network. Each GPU CU, which is analogous to an NVIDIA Streaming Multiprocessor (SM), has a separate node on the network. We believe that this design point better represents the needs of future systems than today's integrated CPU-GPU systems. All CPU and GPU cores have an attached block of SRAM. For CPU cores, this is an L1 cache, while for GPU cores, it is divided into an L1 cache and a scratchpad. Each node also has a bank of the L2 cache, which is shared by all CPU and GPU cores. The stash is located at the same level as the GPU L1 caches and both the cache and stash write their data to the backing L2 cache bank.

Determining a uniform write policy for the L1 caches was a challenge as modern CPUs and GPUs use different policies: CPU multi-core systems commonly use writeback (WB) while GPU CUs use writethrough (WT). To avoid flooding the network with GPU WT requests, modern integrated CPU-GPU systems aggregate all of the GPU's cores at a single node in the network and have a local WB L2 that node to filter the GPU's WT traffic. However, this approach isn't scalable with increasing number of GPU cores. To find an appropriate solution, we considered several choices. We considered adding a shared L3 for both CPU and GPU. However, this wouldn't have resolved the issue of GPU's traffic all emanating from a single node. Instead we decided to make all of the L1 caches in the system use a WB policy. To ensure that the most up-to-date value has been written back to L2, we use a HW-SW co-designed coherence mechanism, as discussed in Section 5.4.4.

6.3 Simulated Memory Configurations

We use an extended version of the DeNovo protocol that supports the stash organization (including our optimizations for data replication). For configurations using scratchpads, only the global memory requests from GPU are seen by the memory system.

To compare the performance of stash against a DMA technique, we enhanced the scratchpad with a DMA engine. Our implementation is based on the D^{2}MA design [70]. D^{2}MA provides DMA capability for scratchpad loads on discrete GPUs and supports strided DMA mappings. Every scratchpad load in D^{2}MA needs to check if it is part of one of the scratchpad blocks that is currently being populated by a pending DMA. When such a check passes, D^{2}MA blocks the execution at a warp granularity. Unlike D^{2}MA, our implementation blocks memory requests at a core granularity, supports DMAs for stores in addition to loads, and runs on a tightly-coupled system.

The DMA optimization for scratchpads gives an additional advantage of prefetching data. To evaluate the effect of prefetching, we applied a prefetch optimization to stash. Unlike DMA for scratchpads, prefetch for stash does not block the core as our stash accesses are globally visible and duplicate requests are handled by the MSHR. We conservatively do not charge additional energy for the DMA or the prefetch engine that issues the requests.

We evaluate the following configurations:

1. $Scratch$: 16 KB Scratchpad + 32KB L1 Cache. The memory accesses use the default memory type specified in the original application.

2. $ScratchG$: $Scratch$ with all global accesses converted to scratchpad accesses.

3. $ScratchGD$: $ScratchG$ configuration with DMA support

4. $Cache$: 32 KB L1 Cache with all scratchpad accesses in the original application converted to global accesses.

5. $Stash$: 16 KB Stash + 32KB L1 Cache. The scratchpad accesses from the $Scratch$ configuration have been converted to stash accesses.

6. $StashG$: $Stash$ with global accesses converted to stash accesses.

7. $StashGP$: $StashG$ configuration with prefetching support.

6.4 Workloads

We present results for a set of benchmark applications as well as four custom microbenchmarks. The larger benchmark applications demonstrate the effectiveness of the stash design on real workloads and evaluate what benefits the stash can provide for existing code. However, these existing applications are tuned for execution on a GPU with current scratchpad designs that do not efficiently support data reuse, control/data dependent memory accesses, and accessing specific fields from an AoS. As a result, modern GPU applications typically do not use these features. But stash is a forward looking memory organization designed both to improve current applications and increase the use cases that can benefit from using scratchpads. Thus, to demonstrate the benefits of the stash, we evaluate it for microbenchmarks designed to show future use cases.

6.4.1 Microbenchmarks

We evaluate four microbenchmarks: Implicit, Pollution, On-demand, and Reuse. Each microbenchmark is designed to emphasize a different benefit of the stash design. All four microbenchmarks use an input array of elements in AoS format; each element in the array is a struct with multiple fields. The GPU kernels access a subset of the structure's fields; the same fields are subsequently accessed by the CPU to demonstrate how the CPU cores and GPU CUs communicate data that is mapped to the stash. We use a single GPU CU for all microbenchmarks. We also parallelize the CPU code across 15 CPU cores to prevent the CPU accesses from dominating execution time. The details of each microbenchmark are discussed below.

Implicit highlights the benefits of the stash's implicit loads and lazy writebacks as described in Section 5.4.2. In this microbenchmark, the stash maps one field from each element in an array of structures. The GPU kernel reads and writes this field from each array element. The CPUs then access this updated data.

Pollution highlights the ability of the stash to avoid cache pollution through its use of implicit loads that bypass the cache. Pollution's kernel reads and writes one field each from two AoS arrays A and B; A is mapped to the stash or scratchpad while B uses the cache. A is sized to prevent reuse in the stash in order to demonstrate the benefits the stash obtains by not polluting the cache. B can fit inside the cache only without pollution from A. Both stash and DMA achieve reuse of B in the cache because they do not pollute the cache with explicit loads and stores.

On-demand highlights the on-demand nature of stash data transfer and is representative of an application with fine-grained sharing or irregular accesses. The On-demand kernel reads and writes only one element out of 32, based on runtime condition. Scratchpad configurations (including ScratchGD) must conservatively load and store every element that may have been accessed. Cache and stash, however, are able to identify a miss and generate a memory request only when necessary.

Reuse highlights the stash's data compaction and global visibility and addressability. This microbenchmark repeatedly invokes a kernel which accesses a single field from each element of a data array. The relevant fields of the data array can fit in the stash but not in the cache because it is compactly stored in the stash. Thus, each subsequent kernel can reuse data that has been loaded into the stash by a previous kernel and lazily written back. In contrast, the scratchpad configurations (including ScratchGD) are unable to exploit reuse because the scratchpad is not globally visible. Cache cannot reuse data because it is not capable of data compaction.

6.4.2 Applications

Table 6.2 lists the seven larger benchmark applications we use to evaluate the effectiveness of the stash. The applications are from Rodinia [41, 43], Parboil [127], and Computer Vision [51, 20].

We manually modified the applications to use a unified shared memory address space (i.e., we removed all explicit copies between the CPU and GPU address spaces present in a loosely-coupled system). We also added the appropriate map calls based on the different stash modes of operation (from Section 5.3.3). The types of mappings used in each application (for all kernels combined) is listed in Table 6.2. The compilation process involved three steps. In the first step, we used NVCC to generate the PTX code for GPU kernels and an intermediate C++ code with CUDA specific annotations and functions. In the second step we edit these function calls so that they can be intercepted by Simics and can be passed on to GPGPUSim during the simulation. This step is automated in our implementation. Finally, we compile the edited C++ files using g++ (version 4.5.2) to generate the binary. We did not introduce any additional compilation overheads compared to a typical compilation of a CUDA program. Even when a CUDA application is compiled for native execution, there are two steps involved - NVCC emitting the PTX code and an annotated C++ program and g++ converting this C++ code into binary (these steps are hidden from the user). All of our benchmark applications execute kernels on 15 GPU CUs. We use only a single CPU core as these applications have very little work performed on the CPU and are not parallelized.

6.5 Results

6.5.1 Access Energy Comparisons

Table 6.1 shows per access hit and miss energies of various hardware components used in our simulations. The table shows that scratchpad access energy (no misses for scratchpad accesses) is 29% of the L1 cache

Application	Input Size	Stash Usage Modes	
		# Mapped Coherent	# Mapped Non-coherent
LUD [41, 43]	256x256 matrix	4	3
SURF [51, 20]	66 KB image	1	1
Backprop [41, 43]	32 KB	2	4
NW [41, 43]	512x512 matrix	2	2
Pathfinder [41, 43]	10 x 100K matrix	1	4
SGEMM [127]	A: 128x96, B: 96x160	1	2
Stencil [127]	128x128x4, 4 iterations	1	3

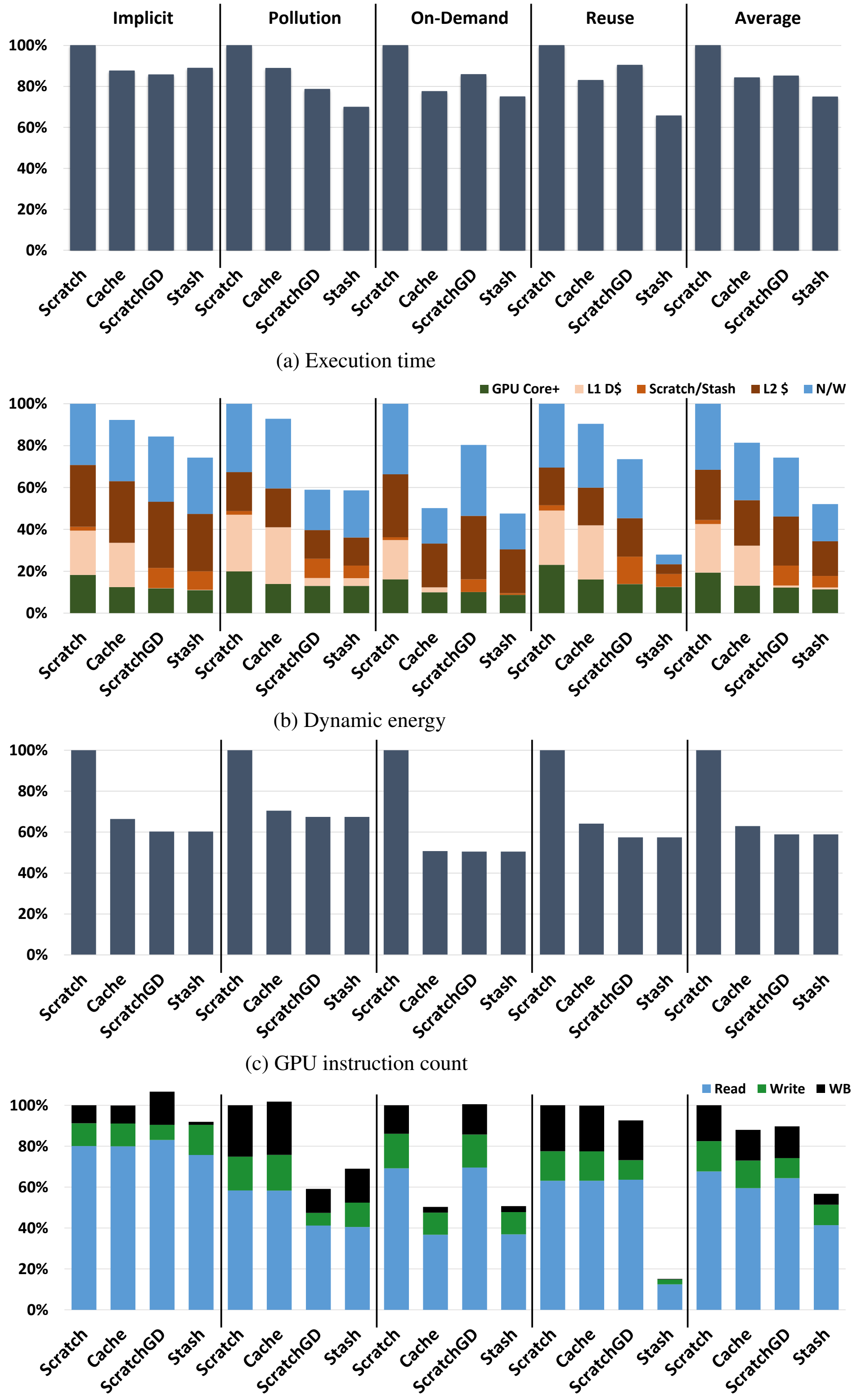
Table 6.2 Input sizes and stash usage modes of Applications.

hit energy. Stash's hit energy is comparable to that of scratchpad and its miss energy is 41% of the L1 cache miss energy. These numbers show that accessing stash is energy-efficient compared to that of a cache and its hit energy is comparable to that of a scratchpad. We use these access energies in our energy evaluations described next.

For L1 caches we assumed an 8-way associative cache with no way prediction. To estimate the difference between the access energy of stash compared to a way-predicted cache, we modeled a similar sized direct-mapped cache and found the hit and miss energies (including TLB access) to be 8.84E-11J and 1.74E-10J respectively. The stash access energies described in Table 6.1 are still less compared to that of a direct-mapped cache, but the difference is much less compared to a 8-way associative cache. If we used a similar sized direct-mapped cache as that of stash (16KB), the hit and miss energies are instead 7.89E-11J and 9.78E-11J respectively, which are still greater than the stash energies. Note that a way misprediction will result in additional energy overheads. As our GPU simulator, GPGPU-Sim, does not support way prediction, we use an associative cache for our energy numbers. If way prediction is used, stash will still outperform in terms of energy per access.

6.5.2 Microbenchmarks

Figure 6.2 shows the execution time, energy, GPU instruction count, and network traffic for our microbenchmarks using scratchpad (Scratch), cache (Cache), scratchpad with DMA (ScratchGD), and stash (Stash). The remaining configurations (ScratchG, StashG, and StashGP) do not aid us in better demonstrating the benefits of stash when used for these microbenchmarks. So we do not evaluate them here. Energy bars are divided by where energy is consumed: GPU core, L1 cache, scratchpad/stash, L2 cache, or network. Network traffic bars are divided by message type: read, write, or writeback.



(a) Execution time

(b) Dynamic energy

(c) GPU instruction count

(d) Network traffic (flit-crossings)

Figure 6.2 Comparison of microbenchmarks. The bars are normalized to the *Scratch* configuration.

CPU Parameters	
CPU Frequency	2 GHz
CPU cores (applications)	1
CPU cores (microbenchmarks)	15
GPU Parameters	
Frequency	700 MHz
GPU cores (applications)	15
GPU cores (microbenchmarks)	1
Scratchpad/Stash Size	16 KB
Number of Banks in Stash/Scratchpad	32
Memory Hierarchy Parameters	
L1 Size (8 banks)	32 KB
L2 Size (16 banks, NUCA)	4 MB
TLB & RTLB	64 entries
Stash map	64 entries
L1 and Stash hit latency	1 cycle
Remote L1 and Stash hit latency	35−83 cycles
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

Table 6.3 Parameters of Simulated Heterogeneous System

Our results show that, on average, the stash is faster and consumes less energy than the scratchpad, cache, and DMA configurations – 25%, 11%, and 12% faster respectively and 48%, 36%, and 30% less energy respectively. Overall, the microbenchmark results show that (a) the stash performs better than scratchpad, caches, and scratchpad with DMA; and (b) data structures and access patterns that are currently unsuitable for scratchpad storage can be efficiently mapped to stash. We next discuss the sources of these benefits for each configuration.

Scratchpad vs. Stash

Compared with the scratchpad configuration, stash enjoys the following benefits of global addressing and global visibility.

Implicit data movement: By implicitly transferring data to local memory, stash executes 40% fewer instructions than scratchpad for the Implicit benchmark and as a result decreases execution time by 11% and energy consumption by 25%.

No cache pollution: Unlike scratchpad, stash does not access the cache when transferring data to or from local memory. By avoiding cache pollution, stash consumes 41% less energy and cuts execution time by 30% in the Pollution benchmark.

On-demand loads into structures: The advantages of on-demand loads can be seen in the On-demand microbenchmark results. Since stash only transfers the data it accesses into local memory, stash reduces energy consumption by 52% and execution time by 25% relative to scratchpad, which must transfer the entire data array to and from the local memory.

Lazy writebacks/Reuse: Lazy writeback enables data reuse across kernels, demonstrated in Reuse. Because it can avoid repeated transfers to and from the same core, stash consumes 72% less energy and executes in 34% less time than scratchpad.

The primary benefit of scratchpad is its energy efficient access. Scratchpad has less hardware overhead than stash and does not require a state check on each access. However, the software overhead required to load and write out data limits the use cases of scratchpad to regular data that is accessed frequently within a kernel. By adding global visibility and global addressability, stash memory eliminates this software overhead and can attain the energy efficiency of scratchpad (and higher) on a much larger class of programs.

Cache vs. Stash

Compared to cache, stash benefits from direct addressability and compact storage. Stash accesses do not need a tag lookup, do not incur conflict misses, and only need to perform address translation on a miss, so a stash access consumes *less energy* than a cache access for both hits and misses. This benefit can be seen in the reduction of the stash energy component of the stash configuration when compared with the L1 D cache energy component of the cache configuration, an average 69% decrease across all microbenchmarks. This contributes to an overall 33% energy reduction on average. The benefits of *data compaction* are demonstrated in the Pollution and Reuse microbenchmarks. The data reuse benefit explored by the Reuse microbenchmark is not applicable to caches. In both cases, the cache configuration is forced to evict and reload data because it is limited by cache line storage granularity and cannot efficiently store a strided array. Stash is able to fit more data in local storage without polluting the cache and achieves a decrease of up to 69% in energy and up to 21% in execution time relative to cache.

Cache is able to store much more irregular structures and is able to address a much larger global data space than stash. However, when a data structure is linearizable in memory and can fit compactly in the stash space, stash can provide much more efficient access than cache with significantly less overhead than scratchpad.

ScratchGD vs. Stash

Applying DMA to a scratchpad configuration mitigates many of the inefficiencies of scratchpad memory by initiating data transfers from hardware and bypassing the cache. Even so, such a configuration still lacks many of the benefits of global addressability and visibility present in stash. First, since scratchpad is not globally addressable, DMA must explicitly transfer all data to and from the scratchpad before and after each kernel. All threads must wait for the entire DMA load to complete before accessing the array, which can stall threads unnecessarily and create bursty traffic in the network. Second, DMA must transfer all data in the mapped array whether or not it is accessed by the program. The *On-demand* microbenchmark highlights the problem when accesses are sparse and unpredictable. Stash achieves 41% lower energy and 50% less network traffic. Third, since scratchpad is not globally visible, DMA is unable to take advantage of *reuse* across kernels; therefore, stash sees 84% traffic reduction, 62% energy reduction, and 27% execution time reduction in the Reuse microbenchmark. DMA also incurs additional local memory accesses compared with stash because it accesses the scratchpad at the DMA load, the program access, and the DMA store. This results in an average 46% reduction in the stash/scratchpad energy component for stash.

Nevertheless, in two cases, DMA does slightly better. First, for Implicit, stash sees 4% higher execution time. While DMA writes back data at the end of each kernel (WB traffic), stash only needs to register modified data (Write traffic). This results in average 33% less network traffic across all microbenchmarks for stash, but the indirection this causes for subsequent remote requests can cause added latency. This is the cause of the slight increase in stash execution time in Implicit. However, this slowdown could be avoided with a prediction mechanism to determine the remote location of data, bypassing the indirection of the registry lookup. The registration requests can also increase traffic if the data is evicted from the stash before its next use, as happens with the pollution microbenchmark. Here the stash suffers higher network traffic because it issues both registration and writeback requests. In general, though, global visibility and addressability improve performance and energy and make stash feasible for a wider range of data access patterns.

These results validate our claim that the stash combines the advantages of scratchpads and caches into a single efficient memory organization. Compared to a scratchpad, the stash is globally addressable and visible; compared to a cache, the stash is directly addressable, has more efficient lookups, and provides compact storage. Compared with a non-blocking DMA engine, the stash is globally addressable and visible and transfers data on-demand, rather than using bulk transfers that conservatively copy all data that may be accessed and can cause unnecessary stalls. As a result, the stash configurations always outperform the scratchpad and cache configurations, even in situations where a scratchpad or DMA engine would not traditionally be used, while also providing decreased network traffic and energy consumption.

6.5.3 Applications

In this section we evaluate the seven configurations on the benchmark applications. First, we compare $Scratch$, $ScratchG$, and $Cache$ configurations. These comparisons are aimed at evaluating the default access pattern (part scratchpad and part global accesses) of the application. Next, we compare $Scratch$ against $Stash$ and $StashG$. With these comparisons, we validate the benefits of stash against scratchpads ($Stash$) and caches ($StashG$). Finally, we evaluate the $ScratchGD$ and $StashGP$ configurations.

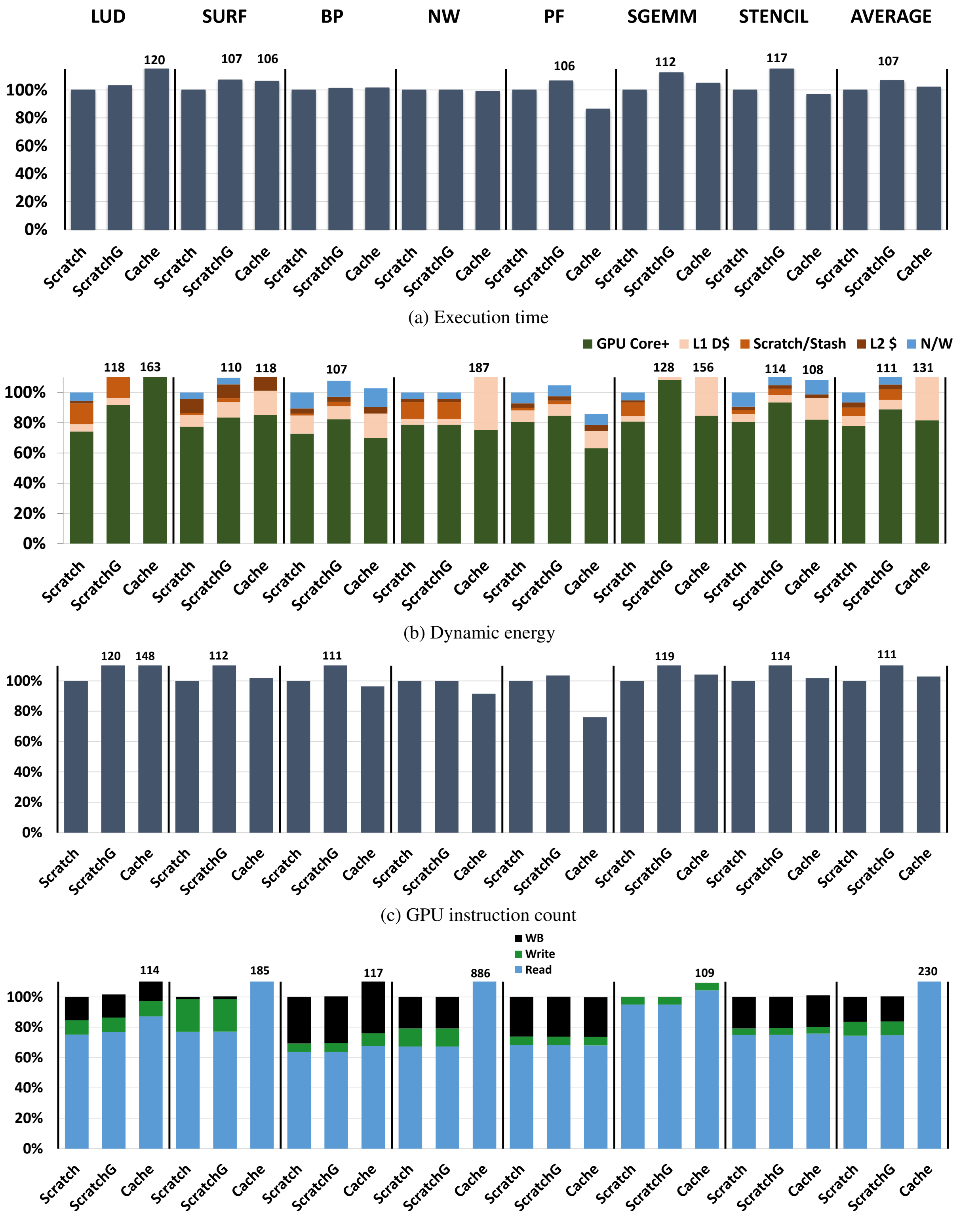
Scratch vs. ScratchG vs. Cache

Figure 6.3 shows the results for execution time, energy consumption, GPU instruction count, and network traffic for the seven applications described in Section 6.4. These applications were selected for their use of scratchpad, and we next focus on comparing the $Scratch$, $ScratchG$, and $Cache$ configurations.

Figure 6.3a shows that $ScratchG$ performs worse compared to $Scratch$ for all applications except for NW (no global accesses in $Scratch$). This is because the global accesses that are converted to scratchpad accesses increase the overall instruction count (the global accesses are better off being global as there is no temporal locality for these accesses). Converting all accesses to global accesses ($Cache$) generally has a slight adverse effect on the execution time, incurring an average execution time increase of 2% relative to $Scratch$. This increase is primarily due to the additional instructions needed to calculate global addresses for applications such as LUD, SURF, and Backprop.

Figure 6.3b shows that $ScratchG$ consumes more energy (up to 29%) as it increases both the instruction count and the number of scratchpad access (the global access converted to scratchpad access still incurs a cache access but also adds a scratchpad access) when compared to $Scratch$. $Cache$ consumes significantly more energy than the other configurations (average 31% more than $Scratch$) because cache accesses are much more power-inefficient than scratchpad or stash accesses. Also cache accesses incur conflict misses resulting in increased network traffic (explained next).

As expected, $ScratchG$ increases instruction count compared to $Scratch$ for all applications (except NW as no global accesses) as more data is accessed via scratchpad (Figure 6.3c). Although $Cache$ does



(a) Execution time

(b) Dynamic energy

(c) GPU instruction count

(d) Network traffic (flit-crossings)

Figure 6.3 Comparing scratchpad and cache configurations for the seven benchmark applications. The bars are normalized to the $Scratch$ configuration.

not need instructions to explicitly transfer data to local memory as $Scratch$ does, converting local accesses to global accesses can introduce additional instructions for index computation in applications like LUD and SURF.

Finally, Figure 6.3d compares the effect of the three configurations on the network traffic. $ScratchG$ performs comparable or slightly worse against $Scratch$. Compared to $ScratchG$, $Cache$ always performs worse as it suffers from conflict misses (more than 8X increase in traffic for NW).

These results show that $Scratch$, the default configuration that the application came with, performs better and is energy-efficient compared to $ScratchG$ and $Cache$ configurations.

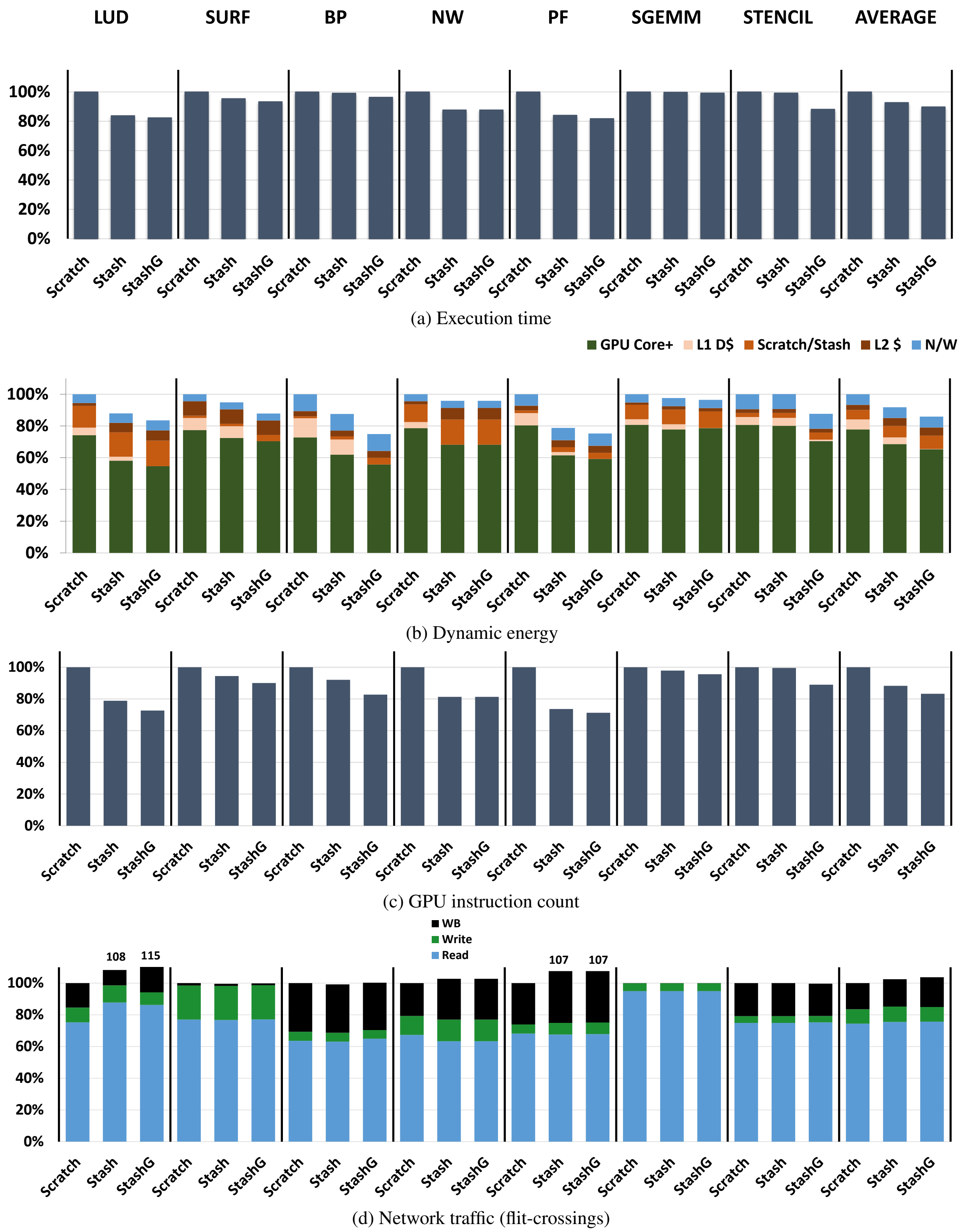
Scratch vs. Stash vs. StashG

Next we evaluate the stash configurations. By comparing $Stash$ and $Scratch$, we show the benefits of stash over a scratchpad. Similarly, by comparing $StashG$ and $Stash$, we show the benefits of stash over a cache.

$Stash$ is up to 20% (average: 8%) faster than $Scratch$ as shown in Figure 6.4a. The improvements are especially good for LUD, NW, and Pathfinder, which exploit the stash's global visibility to reuse data across kernels. The $StashG$ configuration further reduces the instruction count compared to $Stash$ (see below). As a result, using the stash organization for all accesses ($StashG$), we get an overall average reduction of 10% execution time compared to $Scratch$.

Stash configurations show reduction in energy consumption compared to $Scratch$ (Figure 6.4b). This reduction comes from (a) making the stash globally addressable, which removes the explicit copies and decreases both GPU core energy and L1 cache energy ($Stash$ vs. $Scratch$); and (b) converting global accesses to stash accesses ($StashG$ vs. $Stash$). The stash's global addressing also removes cache pollution, which affects the performance for applications like Backprop, and decreases the L1 cache energy compared to the scratchpad. There are two positive energy implications when the global accesses are converted to stash accesses: (i) a stash access is more energy-efficient compared to a cache access; and (ii) many index computations performed by the core for a global access are now performed efficiently by the $Stash - map$ in hardware (seen as reduced 'GPU core+' portion for $StashG$ compared to $Stash$). Overall, $StashG$ reduces energy consumption by up to 25% (average: 14%) compared to $Scratch$.

Figure 6.4c shows that the instruction count reduction for $Stash$ compared to $Scratch$ (12% average, 26% max) is more significant in applications that use the scratchpad/stash heavily such as LUD, NW,



(a) Execution time

(b) Dynamic energy

(c) GPU instruction count

(d) Network traffic (flit-crossings)

Figure 6.4 Evaluating the stash configurations for the benchmark applications. The bars are normalized to the $Scratch$ **configuration.**

Pathfinder, SURF, and Backprop. $StashG$ further reduces the instruction count as mentioned above resulting in an average 17% fewer instructions compared to $Scratch$. Stencil has the most benefit of converting global accesses to stash accesses ($StashG$ vs. $Stash$) in terms of instruction count as there are several data structures that are used as global accesses in the original application. When these global accesses are converted to stash accesses, the number of index computations are reduced and are efficiently performed by the $Stash - map$ in the hardware.

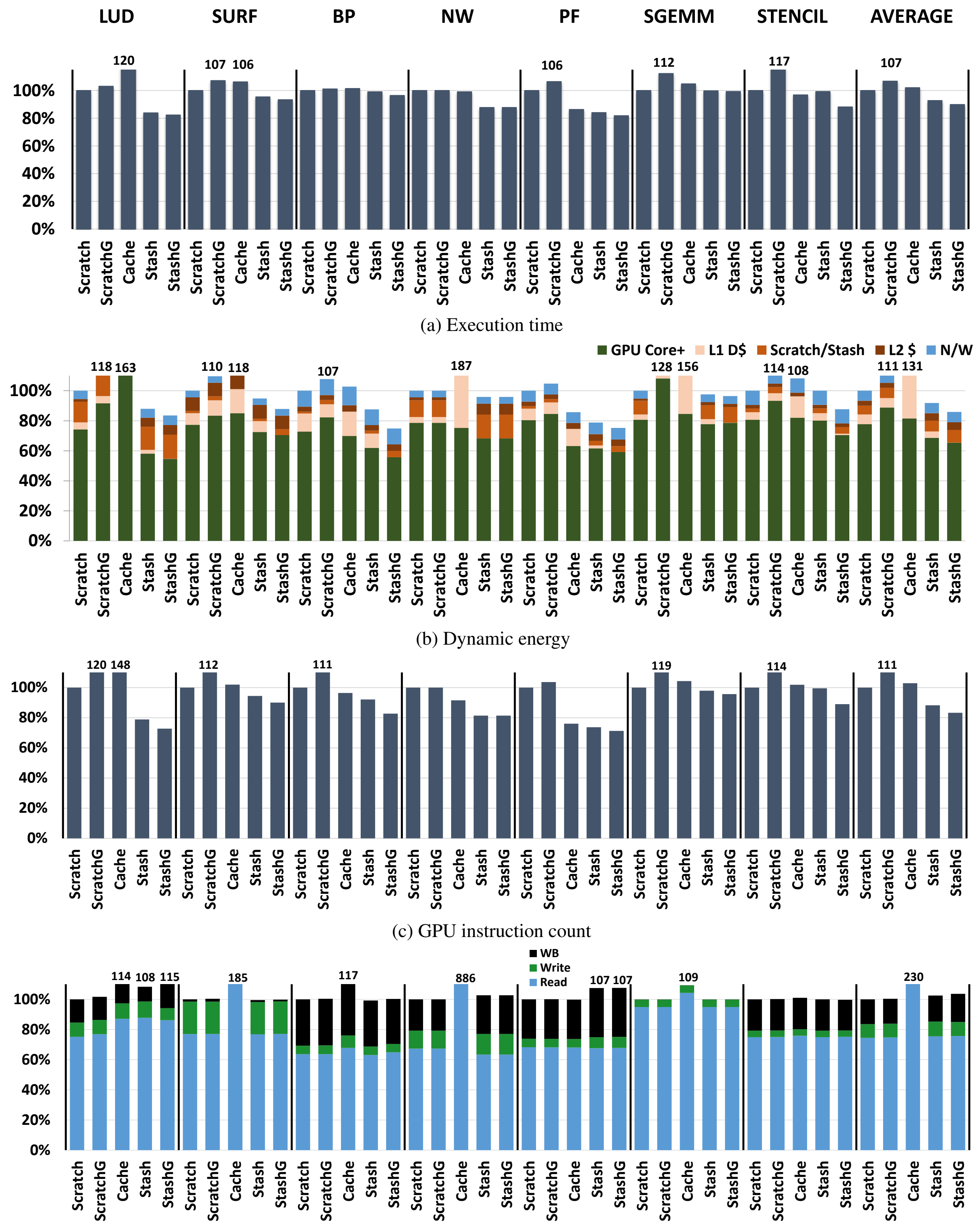
Figure 6.4d shows that stash (both $Stash$ and $StashG$) has little effect on the network traffic for most of the applications compared to scratchpad configurations. The stash configurations show increased writeback traffic for PF. This is due to larger cache size (32KB) compared to the stash size (16KB), which $Scratch$ can utilize to avoid evicting data to the L2. Since stash data bypasses the cache, the larger cache space goes unused in stash configurations. When the stash memory fills up, data is evicted to L2. To validate this, we ran the stash configurations for PF with 48KB stash (keeping the cache at 32KB which is largely unused on the GPU nodes for these configurations) instead and saw that the increased writeback traffic was completely eliminated. So a system that has support for dynamic reconfiguration of on-chip memory can help in this scenario and allocate a larger stash for such applications. The stash configurations also show increased read traffic for LUD. The reason for this behavior is similar to that of the Implicit microbenchmark. As stash keeps the data around, future accesses sometimes result in an indirection through the directory. As mentioned earlier, a prediction mechanism to determine the remote location of data can help in this scenario.

Overall, the stash configurations show that stash has the best of both scratchpad and cache. When both scratchpad and cache accesses are converted to stash accesses, we see a 10% reduction in execution time and 14% reduction in energy consumption on average in our applications.

Figure 6.5 puts the two figures (Figure 6.3 and Figure 6.4) together for a better appreciation of the benefits provided by stash.

StashG vs. ScratchGD vs. StashGP

As mentioned earlier, in addition to avoiding the instruction count overhead, $ScratchGD$ has a potential advantage of prefetching the data compared to $StashG$. The $StashGP$ configuration applies a similar prefetch optimization to $StashG$. As our applications are not written to exploit other benefits that stash provides over the $ScratchGD$ (e.g., on-demand accesses and reuse of data), the performance and energy



(a) Execution time

(b) Dynamic energy

(c) GPU instruction count

(d) Network traffic (flit-crossings)

Figure 6.5 Comparisons of all scratchpad, stash, and cache configurations for the seven benchmark applications. The bars are normalized to the *Scratch* configuration.

differences (small) between $StashG$, $ScratchGD$, and $StashGP$ primarily come from prefetching. The results are mixed though. Applications that have scratchpad/stash accesses not right after the prefetch call points (in the applications we studied, this behavior is seen in all but Stencil), show benefits for $ScratchGD$ compared to $StashG$: 4% reduction in execution time on average. $StashGP$ for these applications show 4.7% reduction on average in execution compared to $StashG$.[3] The slight improvement of $StashGP$ over $ScratchGD$ is attributed to the fact that $StashGP$ does not block the core for pending prefetch requests. Prefetching seems to hurt Stencil. $StashG$ performs better (though the difference is very small) compared to both $ScratchGD$ and $StashGP$ configurations - <1% in cycles against both $ScratchGD$ and $StashGP$. Finally, the difference in energy consumption across the three configurations is negligible (<0.5% on average). None of the three configurations suffer from instruction count overhead. $ScratchGD$ employs DMA to mitigate explicit instructions and $Stash$ and $StashGP$ implicitly move data into the stash. As a result, all the three configurations see the exact same instruction count for all the applications.

6.6 Summary

We evaluate the proposed stash memory organization (Chapter 5) against scratchpad and cache organizations. We use an integrated tightly-coupled CPU-GPU system for our simulations. Using four microbenchmarks, we emphasize the various benefits of stash organization that are not exploited by today's applications. We also provide evaluations for seven larger benchmarks to study how stash performs on applications that exist today. For the larger benchmark applications, compared to the base application ($Scratch$), stash ($StashG$ with global accesses also in stash) shows on average 10% reduction in execution time and 14% reduction in total energy. These results show that even though these applications were not written with the stash in mind, the stash provides substantial energy and performance benefits. Specifically, the $StashG$ configuration shows that the stash organization is more efficient compared to the scratchpad and the cache organizations. Finally, we applied a DMA extension to scratchpad and compared it to a stash configuration with prefetching. These two configurations show similar performance and energy results for the applications studied. Stash provides other benefits compared to DMA but today's applications are not written to exploit these benefits.

[3] Note that the results reported here are against $StashG$ and not against $Scratch$ as we have been discussing so far.

CHAPTER 7

RELATED WORK

In this chapter, we describe the prior work that is related to the proposals made in this thesis. First, we compare our DeNovo coherence protocol against several other techniques that address one or more of the issues targeted by DeNovo. Next, we discuss various techniques available today to verify coherence protocols. Finally, we provide the related work for our stash memory organization.

7.1 Multicore Systems

There is a vast body of work on improving the shared-memory hierarchy, including coherence protocol optimizations (e.g., [83, 95, 96, 115, 128]), relaxed consistency models [55, 58], using coarse-grained (multiple *contiguous* cache lines, also referred to as regions) cache state tracking (e.g., [35, 102, 142]), smart spatial and temporal prefetching (e.g., [124, 137]), bulk transfers (e.g., [12, 40, 64, 66], producer-initiated communication [3, 80]), recent work specifically for multicore hierarchies (e.g., [18, 63, 141]), and many more. The work in this thesis is inspired by much of this literature, but our focus is on a holistic rethinking of the cache hierarchy driven by disciplined software programming models to benefit hardware complexity, performance, and power. Below we elaborate on work that is the most closely related.

The SARC coherence protocol [72] exploits the data-race-free programming model [7], but is based on the conventional directory-based MESI protocol. SARC introduces “tear-off, read-only” (TRO) copies of cache lines for self-invalidation and also uses direct cache-to-cache communication with writer prediction to improve power and performance. Their results, like ours, prove the usefulness of disciplined software for hardware. Unlike DeNovo, SARC does not reduce the directory storage overhead (the sharer list) or reduce protocol complexity. Also, in SARC, all the TRO copies are invalidated at synchronization points while in DeNovo, as shown in Section 4.5, region information and touched bits provide an effective means for selective self-invalidation. Finally, SARC does not explore flexible communication granularity since it does not have the concept of regions and also it is susceptible to false sharing. VIPS-M [118] is an extension to SARC that exploits statically marked private or shared data to perform self-invalidations and write-through for synchronization accesses. VIPS still does not use regions like DeNovo.

Other efforts target one or more of the cache coherence design goals at the expense of other goals. For example, TAP [83] uses self-invalidations but introduces a much more complex protocol. Rigel [75] does not incur complexity but requires traffic-heavy flushing of all dirty lines to the global shared cache at the end of each phase with some assumptions about the programming model. Cohesion, an extension to Rigel, is a hybrid memory model that switches between hardware and software coherence depending on sharing patterns in many-core heterogeneous systems [76]. This work does not address the limitations of either the software or the hardware protocols that it switches between. RegionScout is another compiler-hardware coherence approach [101] that does not support remote cache hits, instead they require writes to a shared-level cache if there is a potential inter-phase dependency. The SWEL protocol [113] and Atomic Coherence [135] work to simplify the protocol at the expense of relying on limited interconnect substrates. SWEL dynamically places read-write shared data in the lowest common level of shared cache and uses a bus for invalidation. Atomic Coherence attempts to simplify the coherence protocol by separating out the races from the protocol. The design avoids the protocol races by requiring each coherence action to be guarded by a mutex. It uses nanophotonics for performing these mutex operations with low latency as they are now on the critical path. As a result, Atomic Coherence eliminates transient states in the coherence protocol but heavily relies on a specific type of on-chip network.

There has also been some work on redesigning the hardware cache coherence protocol to specifically address their verification complexity. A recent proposal, PVCoherence [143], lists various guidelines for designing cache coherence protocols so that they can be verified using existing automatic parametric verification techniques. The verification technique used in the paper, Simple-PV, uses an automatic tool to generate the parametric model and then Murφ to verify the generated model. When the proposed guidelines were applied to the MOESI protocol to make it amenable to parametric verification (in the process also making it even more complex), the authors noticed that the resulting protocol couldn't be verified by Murφ and needed even more changes. The final protocol was verifiable but showed performance degradation (e.g., average 5% and up to 13.8% increase in network traffic). In Fractal Coherence [144, 136], the verification for arbitrary number of cores is made possible by just verifying the minimum system for correctness and verifying that the whole system has fractal behavior. But Fractal Coherence requires addition of states and messages to an existing cache coherence protocol (MOSI as described in [144]) to maintain fractal behavior. Thus it enables verification of the entire system at the expense of increasing the complexity of the base protocol. Also, a specific implementation of Fractal Coherence, TreeFractal, shows a performance degradation ($>$10%) when compared to traditional protocols. FlatFractal [136] proposes techniques to eliminate some of the limitations of TreeFractal, making FlatFractal's performance comparable to the base implementation on 16 cores. But when scaled to 32 cores, FlatFractal shows up to 40% degradation in total execution time. In contrast to the above approaches, DeNovo reduces the complexity of the cache coherence protocol by following a hardware-software co-design approach that eliminates races, thereby reducing the overall verification overhead.

Philosophically, the software distributed shared memory literature is also similar to DeNovo, where the system exploits data-race-freedom to allow large granularity communication (virtual pages) without false sharing (e.g., [6, 22, 38, 27]). These techniques mostly rely on heavyweight mechanisms like virtual memory management, and have struggled to find an appropriate high-level programming model. Recent work [56] reduces performance overheads through hardware support.

Some work has also abandoned cache coherence altogether [67] at the cost of significant programming complexity.

7.2 Verification of Coherence Protocols

This section discusses several existing techniques to verify hardware coherence protocols.

Hardware coherence protocols have numerous transient states and hard-to-cover race conditions making it very difficult to find all the bugs using just simulations or random testing. Hence, formal methods like model checking are often employed to verify their correctness. Model checking is a technique to verify the properties of a system by exhaustive exploration of the state space [47, 114]. McMillan and Schwalbe's seminal work on model checking the Encore Gigamax protocol [97] was the first to apply model checking to verify cache coherence protocols.

Complex systems often exhibit a lot of regularity and symmetry. Ip and Dill developed Murφ [54, 69, 105] which exploits these characteristics by grouping together similar states to verify a reduced state graph instead of the full one. This helps to greatly reduce the amount of time and memory used in verification. Murφ is a widely used tool to formally verify cache coherence protocols; e.g., Sun RMO memory model [109], Sun S3.mp multiprocessor [110], Cray SV2 protocol [5], and Token coherence protocol [34].

We use Murφ for our protocol verification work. Using model checking tools like Murφ for verifying cache coherence protocols is not new per se. We, nevertheless, do discuss some extensions to the canonical coherence protocol modeling technique to model the guarantees provided by disciplined programming languages (namely, data-race-freedom and a disciplined parallel phase behavior). The main contribution of this work is to provide a detailed experience of the verification process of a state-of-the-art publicly available, mature, modern hardware coherence protocol implementation (MESI). We further compare this experience with that of verifying a protocol driven by a hardware-software co-design approach (DeNovo), motivated by simplifying both software and hardware through the same mechanisms (while providing opportunities for performance and energy improvements).

Explicit state exploration model checking tools traditionally have the problem of state space explosion limiting the scalability of such tools. As mentioned in Section 3.2, we ran out of system memory when we increased the verification parameters. We tried a distributed model checker based on Murφ, Preach [24] (similar to Eddy Murphi [99]). But PReach did not help us reduce the number of states explored and hence the memory footprint stayed the same.

There are other verification techniques that do not have the above state explosion problem and can scale to larger systems. Parametric verification [46, 98, 106] and theorem proving [121] are two such techniques. Several of these techniques are sometimes combined together to verify a given system. For example, the verification of Token Coherence [34] is achieved by combining assume-guarantee reasoning and structural induction (in addition to model checking). Even though the techniques employed were able to verify the protocol, in practice, these techniques are either hard for non-specialists to use or error-prone because of laborious manual intervention [143]. There have been proposals to automate parametric verification techniques and minimize manual intervention [46], but such techniques impose severe limitations on the protocols that can be verified. DeNovo, in contrast, is a simpler protocol and makes it feasible to verify with easy to use verification techniques, such as an explicit state model checker. A general survey of various techniques used to verify cache coherence protocols can be found in [111].

7.3 Heterogeneous Systems

7.3.1 Heterogeneous Architectures

There have been numerous tightly coupled heterogeneous system designs in academia and industry. Hechtman and Sorin recently explored the performance benefits of tightly coupled, cache coherent, shared virtual memory heterogeneous systems and found that tightly coupled architectures offer significant potential benefits compared to loosely coupled architectures [65]. HSC [112] uses a region-coherence scheme across CPUs and GPUs to reduce the overhead of providing coherence in integrated CPU-GPU systems.

The AMD-led Heterogeneous System Architecture (HSA) consortium defines an integrating CPU+GPU architecture to enable fast, efficient heterogeneous computing [82]. Data is managed using a single, coherent shared memory with unified addressing. AMD's Fusion [32] and Intel's Haswell [68] architectures are integrated CPU-GPU processors that support "zero-copy"/"InstantAccess" data transfers between CPU and GPU, to allow the CPU and GPU to access each others memory without explicit copies.

Intel's MIC architecture is a more tightly-integrated CPU-GPU heterogeneous system with a single shared memory model [119]. MIC requires program annotations and compiler support to determine which variables will be shared for enabling an efficient coherence protocol.

NVIDIA's Project Echelon integrates CPUs (latency-optimized cores, or LOCs) with GPUs (throughput-oriented cores, or TOCs) together into a tiled architecture with a unified memory hierarchy [73].

Other heterogeneous systems with a fully unified address space are AMD's Berlin processor [117], ARM's Mali and Cortex cores [126], and NVIDIA's Project Denver [53].

The above architectures are all similar to our baseline architecture (a tightly coupled, coherent memory hierarchy with a unified address space but do not distribute their cores on the network). However, the main contribution of our work is making scratchpads globally visible, which is not supported in any of these designs. Moreover, stashes support more fine-grain (chunk-level), and dynamic, choices between coherent caching and private data, than any previous designs.

Runnemede is a heterogeneous Exascale architecture for extreme energy efficiency [37]. It contains a mixture of large cores (CEs) that are latency-oriented, and smaller custom architectures (XEs). Each core has a scratchpad and an incoherent cache, both of which must be managed by the programmer. Like Runnemede, Pangaea uses software to manage a tightly integrated CPU-GPU system [138]. In our work we use a hardware-software co-designed protocol, while Runnemede and Pangaea rely on pure software coherence.

The Cell B.E. processor is composed of a single master CPU (PPE) and multiple accelerators (SPEs) [116]. But the Cell's memory system is loosely coupled and requires explicit data movement between PPE and SPEs.

7.3.2 Improving Private Memories

In this section we discuss the much prior work aimed at improving the private memories for CPUs and GPUs. Table 7.1 summarizes the most closely related work to stash on the basis of the benefits in Table 5.1.

Bypassing L1: (MUBUF [10])):

L1 bypass does not pollute the L1 when transferring data between global memory and the scratchpad, but does not offer any other benefits of the stash. One such example is the MUBUF instruction [10] introduced recently by AMD.

Change Data Layout: (Impulse [36], Dymaxion [42]):

By compacting data that will be accessed together, this technique provides an advantage over conventional caches, but does not explicitly provide other benefits of scratchpads. For example, the Impulse memory controller [36] exploits application specific information to remap memory to shadow virtual and physical addresses providing data compaction in caches and reducing network wastage. A system with Impulse still has other problems with caches such as TLB access, tag comparisons, and conflict misses.

Elide Tag: (TLC [120], TCE [145]):

This technique optimizes conventional caches by removing the need for tag accesses (and TLB accesses for TCE) on hits. Thus, this technique provides some of the benefits scratchpads provide in addition to the benefits of caches. However, it relies on high cache hit rates (which are not common for GPUs) and does not remove conflict misses or provide compact storage of the stash.

Virtual Private Memories (VLS [50], Hybrid Cache [48], BiN [49], Accelerator Store [91]):

Virtual private memories provide many of the benefits of scratchpads and caches. Like scratchpads, they do not require address translation in HW and do not have conflict misses; like a cache, they also reuse data across kernels while avoiding polluting other memories. However, they requires tag checks and incur explicit data movement which prevents lazily writing back data to the global address space. Furthermore, virtual private memories do not support on-demand loads. For example, VLS does on-demand accesses after thread migration on a context switch, but the initial loads into the virtual private store is through DMA.

DMAs: (CudaDMA [19], D^{2}MA [70]):

A non-blocking DMA technique with stride information applied to scratchpad provides two of the benefits of stash, i.e., instruction count reduction for explicit data movement and zero pollution of the L1 cache. D^{2}MA [70] is one such example application of DMA to scratchpad. However, as discussed earlier, such a DMA technique does not provide the benefits of on-demand loads (beneficial with control divergence), lazy writebacks, and reuse across kernels. Finally, a non-blocking DMA scheme as applied in D^{2}MA needs additional checks for consistency, potentially for every load to the scratchpad (e.g., a program where some DMA is always happening), resulting in energy overheads.

Overall, while each of the above techniques provides some of the same benefits as the stash, the key difference is that **none of them provide all of the above benefits**.

7.3.3 Other Related Work

TAP [84] and Staged Memory Scheduling (SMS) [13] look at how to share data between cores on integrated CPU-GPUs. TAP modifies the cache management policy to study its affect on the performance of a GPU application, while SMS modifies the behavior of an integrated CPU-GPUs memory controller to improve system performance and fairness. Both of these projects differ from our work because they focus on scheduling and management policies, while our work focuses on the storage organization unit itself.

Gebhart et al. unify all of the local SRAM blocks in a GPU SM (cache, scratchpad, and register file) [57]. This enables them to dynamically partition the SRAM according to a given application's need and improve performance and energy consumption. While they do not focus on the inefficiencies of the storage organization, we can leverage their techniques in our work.

Singh et al. recently implemented a coherence mechanism for standalone GPUs using a time-based coherence framework that relies on globally synchronized counters to determine when a core is allowed to access a given cache line [122]. Compared to conventional protocols, this reduces overheads significantly by removing invalidations and other coherence transitions such as data races. Our stash functionality is independent of the coherence protocol applied. So we can leverage Singh et al.'s coherence mechanism too. However, our work provides coherence for tightly-coupled, integrated CPU-GPU systems instead of standalone GPUs. Additionally, it does not require global synchronization counters to enforce coherence.

Scratchpads vs. Caches:

Chao et al. recently compared the performance of GPU scratchpad and cache implementations [87] and found that using the scratchpad can significantly improve performance and energy efficiency. The tradeoffs they identify are identical to those that we point out in Section 5.1, however we also introduce a new memory organization, stash, that combines the advantages of caches and scratchpads.

Feature	Benefit	Bypass L1 [10]	Change Data Layout [36, 42]	Elide Tag [120, 145]	Virtual Private Mems [48, 49, 50, 91]	DMAs [19, 70]	Stash
Directly addressed	No address translation HW access	✓	✗	✗, ✓	✓	✓	✓ (on hits)
	No tag access	✓	✗	✓ (on hits)	✗	✓	✓
	No conflict misses	✓	✗	✗	✓	✓	✓
Compact storage	Efficient use of SRAM storage	✓	✓	✗	✓	✓	✓
Global addressing	Implicit data movement	✗	✓	✓	✗	✗	✓
	No pollution of other memories	✓	✓	✓	✓	✓	✓
	On-demand loads into structure	✗	✓	✓	✗	✗	✓
Global visibility	Lazy writebacks to global AS	✗	✓	✓	✗	✗	✓
	Reuse across kernels or phases	✗	✓	✓	Partial	✗	✓
Applied to GPU		✓	✗, ✓	✗	✗, ✗, ✗, ✓	✓	✓

Table 7.1 Comparison of stash and prior work.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

Energy efficiency has become extremely important in designing future computer systems. One of the key consumers of today's on-chip energy is the memory hierarchy. Memory hierarchies in today's systems suffer from several inefficiencies. Many of these inefficiencies are a result of today's software oblivious hardware design. So in this thesis, we focus on three sources of these inefficiencies. We explore information available in the software and propose solutions to mitigate these inefficiencies. Next, we briefly introduce the three inefficiencies and summarize our proposals to address them.

1. **Coherence:** Today's directory based hardware coherence protocols are extremely complex with subtle races and transient states. They also incur overheads in storage for maintaining sharer lists and network traffic for sending invalidation and acknowledgement messages. We propose DeNovo for a class of programs that are deterministic. DeNovo assumes a disciplined programming environment where programs have a structured parallel control; are data-race free and deterministic; and provide information on what data is accessed when. The data-race-freedom guarantee from the software helps eliminate the subtle races and hence the transient states in the coherence protocol. The structure parallel control and the data access information enables DeNovo to employ self-invalidations instead of hardware triggered invalidations messages. As a result, DeNovo is a much simpler protocol (has 15X fewer reachable states compared to that of a state-of-the-art implementation of the MESI protocol when modeled on a model checker.) and incurs no directory storage overhead, no invalidation traffic and no false sharing.

2. **Communication:** The next source of inefficiencies is related to how data is communicated in the

system. Specifically, we focus on the on-chip network traffic originating from and destined to the L1 cache. We add two optimizations to DeNovo to address these inefficiencies. The first optimization is 'Direct cache-to-cache transfer', where data in the remote cache can be sent to the requestor without an indirection to the directory. The second optimization is aimed at reducing the wastage of network traffic destined to the L1 cache. In this optimization, we exploit the software information to smartly transfer the relevant useful data instead of always transferring fixed cache lines. We implemented these optimizations as extensions to the DeNovo protocol. We show that these optimizations did not require any new states (or transient states) in the protocol showing the extensibility of the DeNovo protocol. Including the two communication optimizations, DeNovo reduces the memory stall time by 32% and overall network traffic by 36% on average compared to a state-of-the-art implementation of MESI.

3. **Storage:** There are several organization units to store on-chip data. Caches and scratchpads are two of the popular ones. These two organization units have their own advantages and disadvantages. Caches are transparent to the programmer and are easy to use. But they are power-inefficient. Scratchpads provide compact data storage and predictable data accesses. But their lack of global visibility results in several overheads. We propose $stash$, a new memory organization unit that has the best of the both caches and scratchpads. But stash is futuristic. Today's applications are very much tailored towards how scratchpads or caches are used today. Even though we see benefits on having a stash instead of a scratchpad in today's GPU applications, these applications could be written differently (e.g., exploit reuse across kernels) to exploit the benefits of stash that a scratchpad or a cache does not provide on their own. So we also emphasize the benefits of stash using four microbenchmarks. Compared to a baseline configuration that has both scratchpad and cache accesses, we show that the stash configuration, in which scratchpad and cache accesses are converted to stash accesses, reduces the execution time by 10% and the energy consumption by 14% on average for the applications studied.

8.2 Future Work

Next, we describe some of the future directions for the work presented in this thesis.

8.2.1 Coherence

Beyond Deterministic Codes:

Chapter 2 describes a simple hardware-software co-designed protocol that addresses many of the inefficiencies with today's hardware based, software agnostic directory coherence protocols. We first start off with applying this simple and extendable protocol to multicore systems. We later show in Chapter 5 that it fits perfectly well with heterogeneous systems and can also support innovative memory organizations such as stash. But so far we have focused only on deterministic codes. There has been work on extending the DeNovo protocol with simple extensions with no additional overhead to the protocol itself to support disciplined non-determinism [130]. There is also ongoing work [129] in our research group aimed at supporting arbitrary synchronization patterns. All of this work is focused on multicore systems. Moving forward, we would like to explore these extensions to DeNovo and extend it to heterogeneous systems along with the stash memory organization to efficiently support non-determinism on heterogeneous systems.

8.2.2 Network Traffic

The two network traffic optimizations discussed in Section 2.6 focus on the two different categories of data (used and unused) but are focused only on the traffic destined to L1. There are several inefficiencies with the rest of the on-chip and off-chip traffic too. Below we discuss future work in this space.

Towards efficient on-chip network traffic - going beyond L1:

To remove further inefficiencies from the on-chip traffic, we need to focus on the network traffic originating from and destined to the last level cache (LLC) or the L2. Some of the sources of inefficiencies of L2 network traffic are: sending the entire cache line as part of a writeback even though only a part of the line is modified, sending cache lines to L2 that are read only once by the program (e.g., streaming data), request messages to L2 when it is known for sure that L2 wouldn't have the data, and so on. There has been work in this space in our research group which focused on network traffic inefficiencies at L2 which extended the DeNovo protocol discussed in Chapter 2 [123]. The focus of this work was primarily on multicore systems. Now that we have explored inefficiencies in heterogeneous systems with different memory organizations, we would like to explore the applicability of these optimizations on heterogeneous systems. Moreover, heterogeneous systems open up more opportunities to reduce network traffic inefficiencies. For example, most of the GPU applications divide the data into many chunks and a group of threads (a threadblock) work on each chunk of data. Let's say that this chunk of data is brought into the local scratchpad (or a stash) for performing the computation. Depending on the nature of the given application, it is most likely that this chunk of data needs to be written back to L2 to make space for the next threadblock's chunk of data. So performing DeNovo's "cache line" level registration requests is unnecessary. These registration requests can as well be fused with the writeback message to the L2. Even better, if the total data size the application is going to work on far exceeds the space in L2, the writeback can even bypass L2 all the way to the memory.

Efficient off-chip network traffic:

Traditionally, the data transfer between the memory controller and the DRAM is at a fixed cache-line granularity. The same problems a fixed transfer granularity causes on-chip are caused off-chip too. If the on-chip memory hierarchy doesn't need some of the parts of the cache line then transferring that data from and to DRAM is a clear waste of energy. The DRAM could rather send some other data that is useful for the application. We need to explore different DRAM organizations (e.g. emerging research on die-stacked memory [89]) to enable fine-grained data transfer from and to DRAM.

8.2.3 Storage

Stash at the L1 level for CPUs:

In this thesis, we primarily focused on addressing storage inefficiencies in the context of heterogeneous systems. In the future, we would like to explore the possibility of a stash memory organization for CPUs. We envision that the hardware support would be similar to that of what we proposed for GPUs. The challenge will be in the software interface as today's CPUs do not use any directly addressable memory unit and in effectively mapping chunks of global data to stash.

Stash as a Replacement for Last Level Caches:

Other work from the DeNovo project classifies on-chip waste and focuses on its implications on network traffic [123]. This work did not focus on the storage implications of on-chip waste. The stash memory organization proposed in Chapter 5 is a first step towards addressing this concern. Our focus has primarily been on the core's private memory (e.g., scratchpad or L1 cache). The last level cache too has similar inefficiencies related to data storage. We would like to extend the waste analysis proposed in [123] to heterogeneous systems and quantify the amount of on-chip storage waste, especially at the last level cache. We would like to use these findings to motivate using a stash organization in addition to the last level cache.

Going Beyond Scratchpads and Caches:

Caches and scratchpads are two of the widely used memory organizations in today's memory hierarchies. In this thesis, we have looked at several inefficiencies with caches and scratchpads related to coherence, storage, and communication. But there are other memory organizations that could have other inefficiencies too, such as stream buffers. We would like to explore the inefficiencies with the other memory organizations and study how we can use stash and exploit its strengths to provide a unifying baseline organization. Specifically, we would like to explore the theme that is common across all the work of this thesis - exploit information from the software to mitigate the inefficiencies in the hardware.

REFERENCES

[1] *Workshop on Determinism and Correctness in Parallel Programming, 2009, 2011, 2012, 2013, 2014.*

[2] NVIDIA SDK 3.1. `http://developer.nvidia.com/object/cuda_3_1_downloads.html`.

[3] H. Abdel-Shafi, J. Hall, S.V. Adve, and V.S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 204–215, Feb 1997.

[4] D. Abts, S. Scott, and D.J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *IPDPS*, page 11.2, April 2003.

[5] Dennis Abts, David J. Lilja, and Steve Scott. Toward complexity-effective verification: A case study of the cray sv2 cache coherence protocol. In *In Proceedings of the Workshop on Complexity-Effective Design held in conjunction with the 27th annual Intl Symposium on Computer Architecture (ISCA2000)*, 2000.

[6] Sarita V. Adve, Alan L. Cox, Hya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 26–37, 1996.

[7] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 2–14, May 1990.

[8] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N.K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pages 33–42, 2009.

[9] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *PPoPP*, pages 85–96, 2009.

[10] AMD. Sea Islands Series Instruction Set Architecture. `http://developer.amd.com/wordpress/media/2013/07/AMD_Sea_Islands_Instruction_Set_Architecture.pdf`, February 2013.

[11] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. Sharc: Checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 149–158, New York, NY, USA, 2008. ACM.

[12] R.H. Arpaci, D.E. Culler, A. Krishnamurthy, S.G. Steinberg, and K. Yelick. Empirical evaluation of the cray-t3d: a compiler perspective. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 320–331, June 1995.

[13] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 416–427, Washington, DC, USA, 2012. IEEE Computer Society.

[14] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, November 2002.

[15] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS 2009, pages 163–174, April 2009.

[16] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[17] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society.

[18] Arkaprava Basu, Nevin Kirman, Meyrem Kirman, Mainak Chaudhuri, and Jose Martinez. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *MICRO*, 2007.

[19] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[20] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded Up Robust Features. In Ale Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision ECCV 2006*, volume 3951 of *Lecture Notes in Computer Science*, pages 404–417. Springer Berlin Heidelberg, 2006.

[21] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

[22] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report TR CMU-CS-91-170, CMU, 1991.

[23] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Jan. 2011.

[24] Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial strength distributed explicit state model checking. In *Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*, PDMC-HIBI '10, pages 28–36, Washington, DC, USA, 2010. IEEE Computer Society.

[25] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[26] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[27] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 142–153, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[28] Robert L. Bocchino, Jr. et al. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, pages 97–116, 2009.

[29] Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *38th Symposium on Principles of Programming Languages*, POPL, 2011.

[30] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[31] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[32] Pierre Boudier and Graham Sellers. MEMORY SYSTEM ON FUSION APUS: The Benefits of Zero Copy. *AMD Fusion Developer Summit*, 2011.

[33] Z. Budimlic et al. Multi-core Implementations of the Concurrent Collections Programming Model. In *IWCPC*, 2009.

[34] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Verifying safety of a token coherence implementation by parametric compositional refinement. In *In Proceedings of VMCAI*, 2005.

[35] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 246–257, June 2005.

[36] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 70–, Washington, DC, USA, 1999. IEEE Computer Society.

[37] N.P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R.A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A.K. Mishra, W.R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An Architecture for Ubiquitous High-Performance Computing. In *19th International Symposium on High Performance Computer Architecture*, HPCA, pages 198–209, 2013.

[38] M. Castro et al. Efficient and flexible object sharing. Technical report, IST - INESC, Portugal, July 1995.

[39] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[40] Rohit Chandra et al. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *ICS*, 1994.

[41] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54, 2009.

[42] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM.

[43] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization*, IISWC '10, pages 1–11, 2010.

[44] Byn Choi et al. Parallel SAH k-D Tree Construction. In *High Performance Graphics (HPG)*, 2010.

[45] Byn Choi, R. Komuravelli, Hyojin Sung, R. Smolinski, N. Honarmand, S.V. Adve, V.S. Adve, N.P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2011, pages 155–166, 2011.

[46] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398, 2004.

[47] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[48] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. BiN: A buffer-in-NUCA Scheme for Accelerator-rich CMPs. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 225–230, New York, NY, USA, 2012. ACM.

[49] Jason Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman, and Yi Zou. An energy-efficient adaptive hybrid cache. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 67–72, Piscataway, NJ, USA, 2011. IEEE Press.

[50] Henry Cook, Krste Asanovic, and David A. Patterson. Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments. Technical report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2009.

[51] Michael Cowgill. Speeded Up Robustness Features (SURF), December 2009.

[52] Stephen Curial, Peng Zhao, Jose Nelson Amaral, Yaoqing Gao, Shimin Cui, Raul Silvera, and Roch Archambault. MPADS: Memory-Pooling-Assisted Data Splitting. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 101–110, 2008.

[53] Bill Dally. "Project Denver" Processor to Usher in New Era of Computing. `http://blogs.nvidia.com/blog/2011/01/05/project-denver-processor-to-usher-in-new-era-of-computing/`, January 2011.

[54] David L. Dill et al. Protocol Verification as a Hardware Design Aid. In *ICCD '92*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[55] Michel Dubois et al. Delayed Consistency and its Effects on the Miss Rate of Parallel Programs. In *SC*, pages 197–206, 1991.

[56] Christian Fensch and Marcelo Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *HPCA*, 2008.

[57] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 96–106, Washington, DC, USA, 2012. IEEE Computer Society.

[58] Kourosh Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, pages 15–26, May 1990.

[59] Anwar Ghuloum et al. Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures. Intel White Paper, 2007.

[60] Stein Gjessing et al. Formal specification and verification of sci cache coherence: The top layers. October 1989.

[61] Niklas Gustafsson. Axum: Language Overview. Microsoft Language Specification, 2009.

[62] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, pages 413–422. IEEE, 2009.

[63] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 184–195, 2009.

[64] Kenichi Hayashi et al. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. In *ASPLOS*, pages 196–207, 1994.

[65] Blake A. Hechtman and Daniel J. Sorin. Evaluating Cache Coherent Shared Virtual-Memory for Heterogeneous Multicore Chips. Technical report, Duke University Department of Electrical and Computer Engineering, 2013.

[66] John Heinlein et al. Coherent Block Data Transfer in the FLASH Multiprocessor. In *ISPP*, pages 18–27, 1997.

[67] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *2010 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108–109, 2010.

[68] IntelPR. Intel Delivers New Range of Developer Tools for Gaming, Media. *Intel Newsroom*, 2013.

[69] C.N. Ip and D.L. Dill. Efficient verification of symmetric concurrent systems. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings., 1993 IEEE International Conference on*, pages 230 –234, October 1993.

[70] D. Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke. D2ma: Accelerating coarse-grained data transfer for gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 431–442, New York, NY, USA, 2014. ACM.

[71] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP*, pages 179–188, 1995.

[72] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54 –65, Sept.-Oct. 2010.

[73] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[74] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, pages 13–21, 1992.

[75] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *ISCA*, 2009.

[76] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Cohesion: A Hybrid Memory Model for Accelerators. In *Proceedings of the 37th annual International Symposium on Computer Architecture*, ISCA '10, pages 429–440, New York, NY, USA, 2010. ACM.

[77] Fredrik Kjolstad, Torsten Hoefler, and Marc Snir. Automatic datatype generation and optimization. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 327–328, New York, NY, USA, 2012. ACM.

[78] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. Revisiting the complexity of hardware cache coherence and some implications. In *To appear in TACO*, December 2014.

[79] Rakesh Komuravelli, Matthew D. Sinclair, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. Stash: Have your scratchpad and cache it too. In *Submission*.

[80] D. A. Koufaty et al. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *SC*, pages 255–264, 1995.

[81] M. Kulkarni et al. Optimistic Parallelism Requires Abstractions. In *PLDI*, pages 211–222, 2007.

[82] George Kyriazis. Heterogeneous System Architecture: A Technical Review. 2012.

[83] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA*, pages 48–59, Jun 1995.

[84] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *18th International Symposium on High Performance Computer Architecture*, HPCA, pages 1–12, 2012.

[85] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM.

[86] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[87] Chao Li, Yi Yang, Dai Hongwen, Yan Shengen, Frank Mueller, and Huiyang Zhou. Understanding the Tradeoffs Between Software-Managed vs. Hardware-Managed Caches in GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '14, pages 231–242, 2014.

[88] Sheng Li, Jung-Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, pages 469–480, Dec 2009.

[89] Gabriel H. Loh, Nuwan Jayasena, Jaewoong Chung, Steven K. Reinhardt, J. Michael OConnor, and Kevin McGrath. Challenges in heterogeneous die-stacked and off-chip memory systems. In *SHAW-3*, February 2012.

[90] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[91] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The Accelerator Store: A Shared Memory Framework for Accelerator-Based Systems. *ACM Trans. Archit. Code Optim.*, 8(4):48:1–48:22, January 2012.

[92] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[93] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL*, 2005.

[94] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[95] M.M.K. Martin, P.J. Harper, D.J. Sorin, M.D. Hill, and D.A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *Proceedings of 30th Annual International Symposium on Computer Architecture*, ISCA, 2003.

[96] M.M.K. Martin, M.D. Hill, and D.A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings. 30th Annual International Symposium on Computer Architecture*, ISCA '03, 2003.

[97] K. L. McMillan and Schwalbe J. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Conference on Parallel and Distributed Computing*, pages 242–251, Tokyo, Japan, 1991. Information Processing Society.

[98] Kenneth L. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '01, pages 179–195, London, UK, 2001. Springer-Verlag.

[99] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. *Int. J. Softw. Tools Technol. Transf.*, 11(1):13–25, January 2009.

[100] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proc. Fourth Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[101] Sang Lyul Min and Jean-Loup Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):25–44, January 1992.

[102] Andrea Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *ISCA*, 2005.

[103] A.K. Nanda and L.N. Bhuyan. A formal specification and verification technique for cache coherence protocols. In *ICPP*, pages I22–I26, 1992.

[104] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *ACM Trans. Embed. Comput. Syst.*, 8(3):21:1–21:32, April 2009.

[105] C. Norris IP and David L. Dill. Better Verification Through Symmetry. volume 9, pages 41–75. Springer Netherlands, 1996. 10.1007/BF00625968.

[106] J. O'Leary, M. Talupur, and M.R. Tuttle. Protocol verification using flows: An industrial experience. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 172–179. IEEE, 2009.

[107] Marek Olszewski et al. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, pages 97–108, 2009.

[108] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[109] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for rmo (relaxed memory order). In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 34–41, New York, NY, USA, 1995. ACM.

[110] Fong Pong, Michael Browne, Andreas Nowatzyk, Michel Dubois, and Günes Aybay. Design verification of the s3.mp cache-coherent shared-memory system. *IEEE Trans. Comput.*, 47:135–140, January 1998.

[111] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29:82–126, March 1997.

[112] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 457–467, New York, NY, USA, 2013. ACM.

[113] Seth H. Pugsley, Josef B. Spjut, David W. Nellans, and Rajeev Balasubramonian. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, 2010.

[114] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[115] Arun Raghavan, Colin Blundell, and Milo M. K. Martin. Token Tenure: PATCHing Token Counting Using Directory-based Cache Coherence. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 47–58, Washington, DC, USA, 2008. IEEE Computer Society.

[116] M.W. Riley, J.D. Warnock, and D.F. Wendel. Cell Broadband Engine Processor: Design and Implementation. *IBM Journal of Research and Development*, 51(5):545–557, 2007.

[117] Phil Rogers, Joe Macri, and Sasa Marinkovic. AMD heterogeneous Uniform Memory Access (hUMA). AMD, April 2013.

[118] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 241–252, New York, NY, USA, 2012. ACM.

[119] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming Model for a Heterogeneous x86 Platform. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, pages 431–440, New York, NY, USA, 2009. ACM.

[120] Andreas Sembrant, Erik Hagersten, and David Black-Shaffer. TLC: A Tag-less Cache for Reducing Dynamic First Level Cache Energy. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 49–61, New York, NY, USA, 2013. ACM.

[121] Park Seungjoon and Dill David. Verification of flash cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 288–296, New York, NY, USA, 1996. ACM.

[122] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache Coherence for GPU Architectures. In *19th International Symposium on High Performance Computer Architecture*, HPCA 2013, pages 578–590, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[123] Robert Smolinski. Eliminating on-chip traffic waste: Are we there yet? Master's thesis, University of Illinois at Urbana-Champaign, 2013.

[124] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 252–263, 2006.

[125] D.J. Sorin, M. Plakal, A.E. Condon, M.D. Hill, M.M.K. Martin, and D.A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, 2002.

[126] Steve Steele. ARM GPUs: Now and in the Future. `http://www.arm.com/files/event/8_Steve_Steele_ARM_GPUs_Now_and_in_the_Future.pdf`, June 2011.

[127] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and WMW Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012.

[128] Karin Strauss, Xiaowei Shen, and Josep Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 327–338, 2006.

[129] Hyojin Sung and Sarita V. Adve. Supporting Arbitrary Sychronization without Writer-Initiated Invalidations. In *To appear in ASPLOS*, 2015.

[130] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 13–26, New York, NY, USA, 2013. ACM.

[131] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. Denovond: Efficient hardware for disciplined nondeterminism. *Micro, IEEE*, 34(3):138–148, May 2014.

[132] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 276–286, New York, NY, USA, 2003. ACM.

[133] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[134] Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. Inferring method effect summaries for nested heap regions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 421–432, Washington, DC, USA, 2009. IEEE Computer Society.

[135] D. Vantrease, M.H. Lipasti, and N. Binkert. Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In *Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[136] Gwendolyn Voskuilen and T.N. Vijaykumar. High-performance fractal coherence. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 701–714, New York, NY, USA, 2014. ACM.

[137] T.F. Wenisch, S. Somogyi, N. Hardavellas, Jangwoo Kim, A. Ailamaki, and Babak Falsafi. Temporal Streaming of Shared Memory. In *Proceedings of 32nd International Symposium on Computer Architecture*, ISCA '05, pages 222–233, 2005.

[138] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang. Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 52–61, New York, NY, USA, 2008. ACM.

[139] Steven Cameron Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[140] David A. Wood et al. Verifying a multiprocessor cache controller using random case generation. *IEEE DToC*, 7(4), 1990.

[141] J. Zebchuk, M.K. Qureshi, V. Srinivasan, and A. Moshovos. A Tagless Coherence Directory. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2009.

[142] Jason Zebchuk, Elham Safi, and Andreas Moshovos. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In *MICRO*, pages 314–327, 2007.

[143] Meng Zhang, Jesse D Bingham, John Erickson, and Daniel J Sorin. PVCoherence : Designing Flat Coherence Protocols for Scalable Verification. In *Proceedings of the 2014 IEEE International Symposium on High Performance Computer Architecture*, pages 1–12, 2014.

[144] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 471–482, Washington, DC, USA, 2010. IEEE Computer Society.

[145] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Tag Check Elision. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pages 351–356, New York, NY, USA, 2014. ACM.