© 2010 John Henry Kelm

HYBRID COHERENCE FOR SCALABLE MULTICORE ARCHITECTURES

BY

JOHN HENRY KELM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Associate Professor Steven S. Lumetta, Chair
Associate Professor Sanjay J. Patel
Matthew I. Frank, Ph.D.
Assistant Professor Deming Chen

ABSTRACT

This dissertation describes a cache architecture and memory model for 1000-core microprocessors. Our approach exploits workload characteristics and programming model assumptions to build a hybrid memory model that incorporates features from both software-managed coherence schemes and hardware-managed cache coherence. The goal is to achieve the scalability found in compute accelerators, which support relaxed ordering of memory operations and programmer-managed coherence, while providing a programming interface that is akin to the strongly ordered cache coherent memory models found in general-purpose multicore processors today.

The research presented in this dissertation supports the following thesis: *To be scalable and programmable, future multicore systems require a cached, single-address space memory hierarchy. A hybrid software and hardware approach to coherence management is required to support such a memory hierarchy in 1000-core processors and is achievable only by leveraging the characteristics of target applications and system software.*

We motivate a hybrid memory model and present our approach to addressing the challenges facing such a model. We discuss and evaluate a scalable 1024-core architecture, workloads that we see as targets for such an architecture, a memory model that relies on software management of coherence, and scalable hardware coherence schemes. Using these components, we develop the software and hardware support for a hybrid memory model. We demonstrate that our techniques can be used to reduce hardware design complexity, to increase software scalability, or to combine the two.

I dedicate this dissertation to my mother for her inspiration as a person and unwaivering support in all my academic, professional, and personal pursuits.

ACKNOWLEDGMENTS

The work presented in this dissertation is the result of the efforts of many individuals. The students and faculty I have interacted with during my time at the University of Illinois have enriched the last five and a half years of my life and I apologize if I make any omission in acknowledging those who supported me during my stay in graduate school.

I thank the Rigel group for their assistance in cultivating the ideas that went into this dissertation. Danny Johnson and Matt Johnson were instrumental in the design and implementation of many aspects of this dissertation. Without Danny's seemingly infinite capacity to take opposing sides of any argument I put forth, I doubt any of our papers would have had the rigor necessary for acceptance to conferences. Matt's work extending various parts of the Rigel infrastructure and his efforts helping to develop WayPoint were key to the success of this dissertation. Neal Crago has always provided helpful feedback on all aspects of my work and has given me an opportunity to help him develop his own dissertation ideas, which has been very rewarding in itself. Aqeel Mahesri, whose in-depth knowledge a variety of information is unmatched, was key to motivating the initial Rigel design. I thank Bill Tuohy for his experienced guidance and the high-quality compiler infrastructure used in this work. Above all, the Rigel team has served as a close group of friends from whom I have many fond memories to take with me beyond graduation.

Advanced Micro Devices has provided material support for my research through

grants and a fellowship in 2007, for which I am thankful. Moreover, my summer spent interning at AMD and the people I met played a key role in my development as a computer architect. In particular, I would like to thank my mentor at AMD, Ben Serebrin, for his patience and guidance that has continued to this day.

I thank my committee members Deming Chen, Matthew Frank, and Sanjay Patel for providing feedback and guidance. In particular, I thank Matt Frank for his advice on bringing a systems project from nothing into a reality. Moreover, if it were not for his urging in 2007 that I take ownership of the initial design documents for Rigel, none of this would have come into being. Sanjay Patel, who provided the original concept of Rigel, has provided a great deal of support in the development of Rigel. His efforts have been critical to the success and visibility of the project. I also want to thank Rose Harris, Marie-Pierre Lassiva-Moulin, and Lila Rhodes for all of their administrative support.

Without the support of my adviser, Steven Lumetta, this dissertation would not be possible. Steve's meticulous attention to detail, mastery of all things technical (and many things arcane), professional and personal advice, and unwaivering pursuit of excellence have improved my work and shaped my world view in innumerable ways. I am deeply indebted to the great patience Steve has shown in my path from a naïve and unfocused first-year graduate student to a where I am today.

Graduate school is as much about personal development as it is professional development. As such, I must thank my many friends for their support during my time in Illinois. My dear friend, David Albrecht, has served as a close confidant and general instigator of trouble. I owe him much gratitude for keeping graduate school interesting. Shane Ryoo provided countless hours of professional advice early on in my career and continues to be one of my closest friends. Greg Colombo, Isaac Gelado, Tom Hughes, David Kaplan, Mark Murphy, and Kirsten Stark have

all supported me professionally and personally during my time in graduate school. I cannot thank Brandon and Melissa Swamy enough for their personal support during my first two years of graduate school and their continued support to this day. Lastly, I thank Bethany Krebs for her love and support during the last year and a half of graduate school. She has brought much joy into my life and given me a different perspective on the world.

I am deeply indebted to my mother Cynthia for always putting my education and upbringing ahead of all other concerns. She has served as a constant source of inspiration and demonstrated true perseverance. Without her support and love this dissertation would not have been possible. Above all else, her unequivocal and unconditional support of my decisions and desires I have come to appreciate more than any other form of support I have ever received.

TABLE OF CONTENTS

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF ABBREVIATIONS xv

CHAPTER 1 Introduction 1

CHAPTER 2 Motivation and Background 10
- 2.1 Trends in Increased Parallelism 10
- 2.2 The Limits of CMP Scalability 12
- 2.3 Elements of Parallel Accelerator Design 15
  - 2.3.1 Element 1: Execution Model 17
  - 2.3.2 Element 2: Work Distribution 18
  - 2.3.3 Element 3: Synchronization 19
  - 2.3.4 Element 4: Locality Management 20
  - 2.3.5 Element 5: Memory Model 21
  - 2.3.6 Low-Level Programming Interface 23
- 2.4 Design Space of Cache Coherence Protocols 25
- 2.5 Discussion and Summary 30

CHAPTER 3 Architecture Baseline 31
- 3.1 Rigel Architecture 31
  - 3.1.1 Overview 31
  - 3.1.2 Cache Management 33
  - 3.1.3 Coherence and Synchronization 37
  - 3.1.4 Scalability 38
- 3.2 Rigel Task Model 39
  - 3.2.1 Software API 40
  - 3.2.2 Queue Management 40
  - 3.2.3 Implementation 42
- 3.3 Feasibility of Physical Design 46
- 3.4 Discussion and Summary 46

CHAPTER 4 Workload Characterization . . . 48
4.1 Workload Description . . . 48
4.2 Bulk-Synchronous Patterns in Accelerator Workloads . . . 53
4.2.1 Parallelism Structure . . . 53
4.2.2 Sharing Patterns . . . 54
4.2.3 Accelerator Workload Characteristics . . . 56
4.2.4 Cache Coherence Management . . . 58
4.3 Coherence Overhead for Accelerator Workloads . . . 58
4.3.1 Network Traffic . . . 59
4.3.2 Software-Managed Coherence Efficiency . . . 60
4.3.3 Performance Trade-Offs in Coherence Management . . . 61
4.4 Discussion and Summary . . . 62

CHAPTER 5 Task-Centric Memory Model . . . 63
5.1 Design . . . 63
5.1.1 Coherence Algorithm . . . 65
5.1.2 Memory Ordering . . . 68
5.2 Optimizations . . . 69
5.3 Coherence Management Placement . . . 72
5.4 Summary and Discussion . . . 76

CHAPTER 6 Scalable Probe Filtering . . . 77
6.1 Baseline Probe Filter Architecture . . . 78
6.2 Scalable Probe Filtering Architecture . . . 79
6.2.1 Broadcast . . . 80
6.2.2 Collection . . . 81
6.3 Summary and Discussion . . . 82

CHAPTER 7 WayPoint . . . 84
7.1 Motivation . . . 84
7.1.1 Sharer Tracking . . . 85
7.1.2 Directory Size . . . 85
7.1.3 Directory Associativity . . . 86
7.1.4 Replacement Policy . . . 88
7.2 Design . . . 90
7.2.1 Directory Coherence Protocol . . . 90
7.2.2 WayPoint Design . . . 91
7.2.3 WayPoint Operation . . . 93
7.2.4 Accessing Entries in the Overflow Directory . . . 93
7.2.5 Release Messages . . . 94
7.2.6 Overflow Directory Removal . . . 95
7.2.7 Hardware Directory Cache Eviction . . . 95
7.3 Summary and Discussion . . . 97
7.3.1 Optimizations . . . 97
7.3.2 Impact of Other Workloads . . . 98

7.4 Summary . . . 99

CHAPTER 8 COHESION . . . 100
8.1 Motivation . . . 100
8.1.1 The Case for Software-Managed Cache Coherence . . . 101
8.1.2 The Case for Hardware Cache Coherence . . . 102
8.1.3 A Hybrid Memory Model . . . 104
8.1.4 Summary . . . 105
8.2 Design . . . 106
8.2.1 Hardware Coherence Protocol . . . 107
8.2.2 Software Coherence Protocol . . . 109
8.2.3 COHESION: A Hybrid Memory Model . . . 110
8.2.4 Software Interface to COHESION . . . 113
8.2.5 Coherence Domain Transitions . . . 114
8.2.6 $HW_{cc} \Rightarrow SW_{cc}$ Transitions . . . 116
8.2.7 $SW_{cc} \Rightarrow HW_{cc}$ Transitions . . . 117
8.3 Protocol Optimizations . . . 119
8.4 Software Use Cases . . . 120
8.5 Programming Examples . . . 123
8.5.1 Static Partitioning . . . 123
8.5.2 Dynamic Partitioning . . . 124
8.5.3 System Software . . . 125
8.6 Compatibility . . . 127
8.7 Summary . . . 129

CHAPTER 9 Evaluation . . . 130
9.1 Methodology . . . 130
9.2 Results: Task-Centric Memory Model . . . 132
9.2.1 Policy Selection . . . 133
9.2.2 Software Coherence Action Utility . . . 135
9.3 Results: Scalable Probe Filtering . . . 137
9.4 Results: WAYPOINT . . . 138
9.4.1 Directory Cache Sizing . . . 138
9.4.2 Directory Cache Associativity . . . 140
9.4.3 Power and Area Estimates . . . 142
9.5 Results: COHESION . . . 143
9.5.1 Message Reduction . . . 143
9.5.2 Directory Entry Savings . . . 144
9.5.3 Directory Area Estimates . . . 146
9.5.4 Application Performance . . . 147
9.5.5 COHESION Summary . . . 148
9.6 Summary and Discussion . . . 149

CHAPTER 10 Related Work 150
10.1 Task-Centric Memory Model Related Work 150
10.1.1 Parallel Programming Models 152
10.1.2 Parallel Memory Models 152
10.1.3 Accelerator Workloads 154
10.2 WayPoint Related Work 154
10.2.1 Coherence Management 154
10.2.2 Directory Cache Associativity 157
10.3 Cohesion Related Work 158
10.3.1 Hardware Schemes 159
10.3.2 Software-Based and Hybrid Schemes 160

CHAPTER 11 Summary and Conclusions 163
11.1 Implementing Coherence in Future Processors 164
11.2 Symmetry Versus Asymmetry 164
11.3 Summary 168
11.3.1 The Task-Centric Memory Model 168
11.3.2 WayPoint: Scalable Hardware Cache Coherence 169
11.3.3 Cohesion: A Hybrid Memory Model for Accelerators . . 170
11.4 Conclusions 170

APPENDIX A Task-Centric Memory Model Formal Specification 172

REFERENCES 179

AUTHOR'S BIOGRAPHY 192

LIST OF TABLES

2.1 Design parameters for coherence on throughput-oriented architectures similar to our baseline. We use the following abbreviations with values used in our evaluation in parentheses: **n**: Potential sharers (128). **S**: Number of active sharers ($0 \leq \mathbf{S} \leq 128$). **i**: Number of sharer pointers in limited schemes (4). **C**: Sharers covered by a coarse-grained vector bit (often 4). $L2_s$: Number of sets per sharer L2 (128). $L2_w$: Number of ways per sharer L2 (16). **M**: Total lines in memory (roughly 128 million for 4GB of memory). **E**: Maximum number of cache lines tracked by on-die structure ($1 \leq \mathbf{E} \leq L2_s \times L2_w \times n$). **F**: Probe filter size. **T**: Tag size. 25

4.1 Characterization of the ten workloads used in this dissertation run using a 1024-core variant of the Rigel architecture with software-managed coherence. 52
4.2 Characterization run in Table 4.1 with an on-die full-directory hardware coherence implementation. 52

5.1 Comparative advantages and disadvantages for different coherence action placements. 75

8.1 Differences in design goals and architectural features between general-purpose CPUs and accelerators such as GPUs. 101
8.2 Trade-offs for HW_{cc}, SW_{cc}, and COHESION. 106
8.3 Programmer-visible software API for COHESION. 113

9.1 Timing parameters for the baseline architecture. 131
9.2 Additional sizing and timing parameters for Rigel with cache coherence. 131
9.3 GDDR5 DRAM memory timings used in our simulations. All units are DRAM cycles assuming 3 GHz DDR at 6.0 Gbps per pin. 132
9.4 Overview of coherence management policies for TCMM. 133
9.5 Power and area estimates for a 2048-entry WAYPOINT implementation. 143

LIST OF FIGURES

1.1 Goal of hybrid coherence: Seamless transitions between hardware-managed and software-managed coherence domains. 8

2.1 Software stack for accelerator architectures. API: application-level programming interface, LPI: low-level programming interface, ISA: instruction set architecture. 17
2.2 Classification of cache coherence schemes. We evaluate sparse directories with full-map (Dir_nNB) and limited (Dir_4B) directory entries. We evaluate probe filtering, broadcast invalidate, and WayPoint directory cache eviction policies. (Figure courtesy of Matt R. Johnson.) 26
2.3 Illustration of the difficulty in selecting a directory replacement policy due to the information disconnect between L2 caches and the directory. 29

3.1 Baseline Rigel Architecture. 32
3.2 Baseline kernel speedup relative to a single eight-core cluster. . . . 39

4.1 Categorization of memory access types in Rigel and VISBench benchmarks. Letters X, Y, and Z represent distinct addresses, and t_1 and t_2 represent distinct tasks. 55
4.2 Relative fraction of memory misses for each access type for VISBench. 56
4.3 Relative fraction of memory misses for each access type for the Rigel benchmarks. 56
4.4 Message count from the L2 cache to the shared L3 normalized to software coherence (SW_{cc}) results. Note that HW_{cc} contains some software flushes as an optimization for data known to be no longer needed in the L2. 59
4.5 Writeback and invalidate efficiency for different L2 cache sizes. An inefficient access is one that is performed by software to a line no longer present in the cache. For all of our other experiments we use a 64 kB cache. 60
4.6 Runtime of idealized hardware and software-managed coherence compared to the best case implementation using the Task-Centric memory model. 61

5.1 State transitions for memory blocks in the Task-Centric Memory Model. Actions include: Local loads (L.LD), local stores (L.ST), global loads (G.LD), global stores (G.ST), write backs to the global cache (WB), and cluster cache invalidates (INV). The † notes states that may cache a block at the cluster cache. The set of tasks sharing a block in state X is denoted by T_X. Any transition absent from the diagram is disallowed by the model. 66
5.2 Logical flow of tasks and coherence actions in the Task-Centric Memory Model. 70
5.3 Choice of coherence action locations using the Task-Centric Memory Model. 73

6.1 Illustration of the difficulty in selecting a directory replacement policy due to the information disconnect between L2 caches and the directory. 78
6.2 Probe filter block diagram. 80

7.1 Time varying set distribution at last-level (L3) shared cache. The figure demonstrates that directory cache thrashing can vary in time and it may shift across sets, implying that over-provisioning of directory cache entries may be required to avoid performance impact due to directory cache conflicts. 87
7.2 General pattern for directory accesses. 89
7.3 Architecture of WayPoint. 92

8.1 Cohesion state diagram. 107
8.2 Cohesion architecture. 110
8.3 Cache state transitions between HW_{cc} and SW_{cc}. 117
8.4 Cache state transitions between SW_{cc} and HW_{cc}. 117
8.5 The static Cohesion pattern demonstrated for a 2D stencil computation. The goal is to place the per-cell private regions in the SW_{cc} domain and only use HW_{cc} domain for a small border region where communication is necessary. 124
8.6 The dynamic Cohesion pattern demonstrated using a two-phase parallel sort on four processor cores. The goal is to use HW_{cc} as-needed during the divide phase and avoid HW_{cc} during periods of independent execution, shown as the serial sort in the figure. When the results are ready, HW_{cc} is used to make the produced data available to consumers. 126

9.1 Runtime for different eviction policies compared to a highly optimistic hardware coherence scheme. Software schemes can be selected on a per-application basis. OmniscientCC corresponds to an implementation with hardware coherence disabled, no coherence traffic for writebacks or invalidates, and an omniscient memory model that provides correct values. 134
9.2 The impact of L2 (cluster) cache sizing on writeback efficiency. . . 135
9.3 The impact of L2 (cluster) cache sizing on invalidate efficiency. . . 135
9.4 Scalability of baseline probe filtering with 2048 entries compared to SPF with broadcast-collective support and an on-die full directory. Perfect speedup would be 128× in this figure. . . . 137
9.5 The runtime normalized to the optimistic hardware cache coherence implementation is shown for different sizes of directory cache. Configurations without WayPoint are shown on the left and those with it on the right. 139
9.6 Runtime of WayPoint-enabled simulations with different associativities with fixed directory size. Results normalized to optimistic hardware cache coherence. Note that we have a one-to-one correspondence between sets and WayPoint lists in (a), resulting in slightly less contention and thus better performance for less-associative on-die caches. 141
9.7 Number of messages sent out of the L2 (cluster) cache. 144
9.8 Runtime with different directory cache sizes (in thousands) for our baseline without Cohesion (a) and with Cohesion (b). Part (c) shows the average and maximum number of directory entries used at runtime out of the total number of entries. 145
9.9 Runtime of Cohesion compared to software-managed coherence (SW_{cc}) and hardware-managed coherence (HW_{cc}) using limited directories with four pointers be entry and full-map directories. We show both optimistic assumptions for HW_{cc}, which provides infinite-sized directory caches, and a large, but realizable 16k-entry directory caches. 147

LIST OF ABBREVIATIONS

API	Application Programming Interface
CMOS	Complementary Metal-Oxide Semiconductor
CMP	Chip Multiprocessor
CPU	Central Processing Unit
DLP	Data-Level Parallelism
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
EIEW	Eager Invalidate, Eager Writeback
EILW	Eager Invalidate, Lazy Writeback
FLOPS	Floating-Point Operations Per Second
FP	Floating-Point
FPGA	Field-Programmable Gate Array
GDDR	Graphics Double Data Rate
GPU	Graphics Processing Unit
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
IP	Internet Protocol
IPC	Instructions Per Cycle
LIEW	Lazy Invalidate, Eager Writeback
LILW	Lazy Invalidate, Lazy Writeback

LPI	Low-Level Programming Interface
MIMD	Multiple-Instruction Multiple-Data
MLP	Memory-Level Parallelism
MSHR	Miss Status Holding Register
PGAS	Partitioned Global Address Space
RISC	Reduced Instruction Set Computer
RTL	Register-Transfer Level
RTM	Rigel Task Model
SIMD	Single-Instruction Multiple-Data
SMVM	Sparse Matrix-Vector Multiply
SoC	System on a Chip
SPMD	Single-Program Multiple-Data
SRAM	Static Random Access Memory
TCP	Transmission Control Protocol
TLP	Thread-Level Parallelism

CHAPTER 1

Introduction

There are two dominant multicore architectures deployed: general-purpose chip multiprocessors (CMPs) with multiple general-purpose cores [1–3] and domain-specific multicore accelerators, such as graphics processing units [4] (GPUs). The current trend in integration is leading toward GPU cores and conventional general-purpose cores coexisting on the same die [3]. The next generation of processors will see the distinction between accelerator and general-purpose cores blur as the highly parallel resources begin to expand their reach from supporting purely graphics workloads to more general forms of parallel acceleration. Future systems are likely to be highly parallel architectures with resources for performing sequential processing, data-parallel compute, and graphics all integrated on the same die and supported by a set of consistent and interrelated architecture abstractions and memory models [5].

Integration of heterogeneous cores offers the sequential performance and flexibility of general-purpose cores complemented by area and power efficiency of accelerator cores for parallel execution. From a systems perspective, integration reduces latency between devices and can increase inter-core bandwidth, albeit potentially at the expense of aggregate off-chip bandwidth provided by a multi-chip implementation. However, the first generation of heterogeneous multicore processors has adopted an interface between GPU and CPU that maintains the host-device divide [6] and forces communication to occur via a bus interface not unlike conventional CPU to discrete GPU interconnect protocols that are preva-

lent today. Moreover, the memory model and runtime software support for the CPU and GPU are sufficiently different such that software developed for one is unlikely to execute efficiently or at all on the other. The implication of increased heterogeneity in compute resources in contemporary architectures is that we must revisit the memory model from a full system perspective and not just in terms of the main processor. Current systems lack a consistent memory model that spans highly parallel accelerators and general-purpose processors. The lack of a consistent software target limits applications' ability to support both accelerators and general-purpose processors. As such we are not yet exploiting the full capabilities of integrating accelerator and general-purpose resources. Moreover, the heterogeneity in the memory model and software support is increasing the complexity faced by software developers already strained by the task of developing parallel software to achieve better performance for their applications.

The recent shift toward multicore designs was due in large part to the reduced utility of increased transistor budgets for improving sequential performance, mostly due to power and design complexity constraints [7]. Until recently, the additional transistors provided by advances in process technologies allowed for microarchitectural advances to increase the performance of microprocessors while leaving the programming and memory models relatively unchanged. As an example, current Intel x86 CPUs still support legacy code execution allowing software written over thirty years ago to continue executing on modern CMPs unmodified [8].

The fact that software applications have a much longer lifespan than the systems that they run on [9], coupled with the fact that the software industry is an order of magnitude larger than the semiconductor industry in terms of spending and revenue, makes the impact of continued advances in silicon manufacturing on the need to alter software a critical economic concern for the computing in-

dustry as a whole. However, the increased design complexity and growing power budget of processors that pursue instruction-level parallelism (ILP) have made performance scaling through microarchitectural innovation alone untenable. Put another way, the rate at which microarchitectural complexity and, by extension, die area increased could not keep pace with the additional area provided by process scaling; microarchitecture advances alone appear to have achieved a roughly 20% increase in performance for industry standard integer CPU benchmarks [10] across the last two generations[1] while Moore's law scaling provides a much larger 2× increase in the transistor budget with each process node. To fill the gap between the performance delivered by sequential microarchitecture techniques and the increased transistor budgets provided by Moore's law scaling, we must investigate scalable techniques that can continue to provide performance per watt and performance per area in a way that is useful to software developers. However, as evidenced by the diminishing returns of Moore's law scaling for sequential code, it is unlikely that these techniques will solely be ILP-driven.

The scalability limitations of ILP processors motivate the integration of multiple cores on a single die [11]. However, continuing to simply integrate *more* ILP cores on-die will provide a poor trade-off between performance, power, area, and complexity for highly parallel workloads that can achieve high per-core efficiency with simpler, more area-efficient cores [12] and may not result in additional performance for commercial applications as more cores are added [13].

Compared to integrating more ILP cores, higher performance and lower power can be achieved by integrating a larger number of throughput-oriented, simpler cores [14] capable of exploiting thread-level parallelism (TLP) and data-level par-

[1] We compare two four-core Intel processors running at 3.2 GHz with 130 watt TDP, both on the same 45 nm process. One is the Penryn-class QX9770 while the other is an i7-965. We use the reference performance numbers for SPECint CPU2006 benchmark runs to estimate performance changes across microarchitectures.

allelism (DLP). Moreover, TLP and DLP abound in scalable workloads [12, 15]. However, supporting only throughput-oriented cores would result in decreased performance for workloads with limited parallelism. A method for supporting both ILP-centric cores and DLP-/TLP-centric cores with a consistent memory model and programming abstraction is the limiting factor for exploiting the advances in integration to take advantage of both CMP-style processing and accelerator-style processing; such a method is the focus of this dissertation.

The benefit of integration is a reduction in off-chip power and reduced packaging costs. Integration also provides lower latency and higher bandwidth between integrated devices. To achieve these benefits, it is sufficient to treat the accelerator resources as subordinate, under the control of a host processor core, similarly to how GPUs are accessed today. However, to fully exploit the programmability benefits of integrated systems, a consistent memory model that extends to both devices must exist. Without a consistent memory model, the cost in terms of programmer effort and time-to-solution of explicit address remapping for pointers, data marshalling, and hiding the cost of copy operations is born directly by the developer. Support for the assertion that a consistent memory model is key to the success of platforms with heterogeneous compute resources comes from industry [5, 16] and academia [17, 18], where research groups are investigating supporting heterogeneous resources under a single address space. Moreover, industry is moving forward with heterogeneous systems that share a single address space. However, it is unclear what the underlying hardware and software support should be.

Processors integrating hundreds of execution units, such as GPUs, have been used as programmable compute accelerators [19]. Accelerator designs available commercially have generally lacked support for cache coherence, requiring the use of multiple address spaces both off-die between host and device and on-die

to achieve good performance using software-managed local memories in lieu of hardware-managed caches. These features are a departure from the conventional single address space, cached, shared memory approach found in general-purpose CMPs available today. The lack of support for the current programming and application ecosystem is a programmability disadvantage that hinders more widespread adoption of accelerator systems. The inherent differences in these memory models interferes with portability across CPUs and GPUs, in terms of both correctness and performance. Moreover, the lack of familiar programming abstractions and memory models will result in forthcoming 100+ core CMPs being ill suited to the needs of programmers. Therefore work must be done to build a scalable memory system to enable future 100–1000 core multiprocessors that provide most of the benefits of contemporary CMPs, while achieving the scalability found in GPUs today.

Our approach to providing a scalable memory model for 1000-core multicore systems is manifold. We develop a lightweight hardware coherence scheme, a scalable software-managed coherence protocol, and the mechanisms necessary to efficiently migrate control of coherence between the two coherence domains, thus enabling what we call a *hybrid memory model*. The goal is to provide hardware coherence as a fall-back mechanism for maintaining a shared single address space to developers, while relying on more scalable software techniques in the common case. A limited form of hardware coherence allows for synchronization, collective, and communication operations to be supported efficiently as needed. Moreover, coherence allows applications to be supported that would otherwise be difficult to map to a software-managed memory model, albeit at some cost in terms of performance. Developers can use the base hardware coherent system to more easily port existing shared memory applications and debug and test new shared memory applications on a hybrid memory model. When possible software, in

the form of either a tools-driven or developer-driven mechanism, can migrate data from the hardware coherence domain to the software-managed domain, thus reducing the pressure on the hardware coherence mechanisms and interconnect using an iterative refinement approach.

Designing and implementing a memory model for a CMP, as is proposed in this work, is a challenging problem because it involves balancing many aspects of the system. Striking a proper balance is confounded by the facts that application characteristics play a large role in the effectiveness of a given memory subsystem and that applications for CMPs are numerous and vary greatly. In fact, the applications that will run on future CMPs with more than a thousand cores likely do not exist yet. Moreover, the ease with which applications can be mapped to a platform, i.e., programmer hours-to-solution, is often at odds with the compute density of the platform, i.e., high $\frac{FLOPS}{mm^2}$. The root of the conflict is the cost of providing latency reducing mechanisms. Examples of latency reducing techniques include out-of-order cores and large caches, which are used to smooth performance cliffs, but do not provide performance benefits commensurate with hardware and complexity costs [20]. What we aim to do is provide limited support for otherwise costly mechanisms, thus reducing initial programming effort, but then provide support for scalable low-cost mechanisms that allow developers concerned with performance to achieve that performance when desired.

The memory system trade-offs have a profound impact on performance, programmability, power dissipation, required verification effort, and design complexity. Furthermore, the memory model is pervasive in that it impacts and is impacted by the choice of network topology, cache sizes, core complexity, and latency tolerance mechanisms, such as multi-threading or out-of-order execution, memory bandwidth, and programming model and language support. The design of a hybrid memory model involves those same trade-offs while adding another di-

mension: a hybrid memory model must trade off hardware overhead and network utilization overhead of hardware coherence for potentially increased software complexity or degraded application performance when software-managed coherence is inefficient. The goal of this dissertation is explore this new dimension and, in doing so, to demonstrate that a hybrid memory model is critical to achieving scalability for 1000-core processors.

We motivate our work by evaluating the design space of coherence architectures and then present a set of design considerations, which we refer to as *elements*, that represent the space of design choices for 1000-core architectures such as Rigel (Chapter 2). We present the design and evaluation of a 1024-core accelerator architecture called Rigel and a task-based programming model called the Rigel Task Model (RTM) that we use as our baseline (Chapter 3). We present the analysis of data- and task-parallel workloads targeting Rigel and show patterns in data access streams and program execution that can be exploited in the design of a scalable memory model (Chapter 4). Building on Rigel, we introduce a novel software-managed coherence scheme that we call the *Task-Centric Memory Model* (TCMM) (Chapter 5). We evaluate techniques for extending the scalability of *probe filtering*, which is a common technique used in CMPs today (Chapter 6). We present a lightweight hardware coherence scheme that we call WayPoint (Chapter 7). The synthesis of hardware cache coherence and a software-managed scheme is investigated as part of Cohesion (Chapter 8) and serves as the baseline for our work on a hybrid memory model. The result is a hybrid coherence scheme that allows designers to trade off hardware complexity and performance without sacrificing the programming model. Hybrid coherence, as shown in Figure 1.1, also enables developers to trade off ease of use for performance by re-partitioning data across coherence domains dynamically.

The scope of a project tasked with developing a new memory model is broad

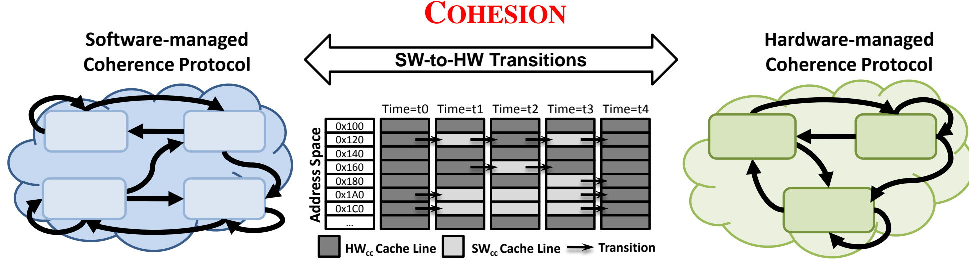


Figure 1.1: Goal of hybrid coherence: Seamless transitions between hardware-managed and software-managed coherence domains.

enough such that addressing all of the issues in a single dissertation is not feasible. We choose to explicitly exclude a set of topics and make what we believe to be reasonable assumptions or projections in lieu of attempting to address those issues. We see the need for more support for software-managed coherence from both the runtime and code generation tools, neither of which are addressed in great detail in this dissertation. Exhaustive verification of the hardware and software protocols found in this dissertation could be performed, but may provide little additional insight and may be infeasible given the current tools' inability to support large designs. For pieces of our design that can be mapped to existing systems, we elide proof of correctness since a similar system has already been realized. Moreover, we adopt existing protocols, such as an MSI directory protocol, and thus obviate the need for proof of correctness while focusing on novel implementation and integration strategies.

The contributions of this work include:

1. A Task-Centric Memory Model: A protocol for maintaining a coherent single address space on a cached processor that uses software-management in lieu of hardware cache coherence.

2. WayPoint: A lightweight hardware coherence scheme that reduces the on-die directory overhead without greatly impacting performance.

3. Cohesion: A set of protocols for dynamically transitioning between hardware-managed and software-managed cache coherence.

4. An evaluation of a hybrid coherent system that allows heterogeneous cores to share a single address space that supports a variety of hardware and software coherence options.

CHAPTER 2

Motivation and Background

In this chapter we motivate the study of hybrid approaches to coherence management for future chip multiprocessors. We discuss the trends in the industry supporting our design choices. We discuss the limitations of current approaches that will confound scalability in the coming generations of CMPs. As a framework for exploring the design space of accelerators, we present a set of design *elements* that we find to be key considerations for building a 1000-core architecture. We conclude by surveying the current approaches to coherence management and form a taxonomy of existing schemes.

2.1 Trends in Increased Parallelism

Prior to the advent of general-purpose chip multiprocessors, parallel architectures were mostly relegated to server-class systems [21]. The introduction of the POWER4 from IBM [22], Intel's Itanium 2 [23] and Smithfield [24], and AMD's Opteron processors [25] marked the start of the current trend of increasing core count in lieu of core frequency. For a single application to exploit the full computational power of a CMP, that application must be parallel. If application performance that improves with newer CMPs is to be expected, those applications must be programmed in a way that exposes greater parallelism than can be exploited at development time. The need for programmers to expose parallelism to achieve performance is a profound departure from the previous model

of sequential software, which took advantage of microarchitectural advances and frequency scaling to continue to provide greater performance without additional burden being placed on the programmer.

The compute accelerator is another class of parallel architecture that is becoming pervasive. Unlike the specialized parallel computers of the past owned only by large businesses, universities, and government labs [21], compute accelerators are available as commodity pieces of hardware. As such, these devices are being programmed by larger numbers of developers. A dominate example of an accelerator is the GPU. GPUs incorporate as many as 1600 processing cores on a single die [26]. The initial developers that tried to exploit the performance potential of GPUs, such as [27], were required to use low-level, graphics-specific APIs such as OpenGL [28]. Eventually higher level tools were developed, such as Brook [29], and today GPUs and multicore CPUs can be programmed using industry standard cross-platform data-parallel languages such as OpenCL [30]. The interest in large-scale parallelism in the consumer market is evidenced by the adoption of accelerator-centric languages at the operating system level [31] and the large number widely used applications that use GPUs outside of graphics and other accelerators to achieve greater performance.

If a greater level of performance is to be achieved from future CMPs, it is clear that parallelism must be exploited. Power is also a key concern for all market segments including mobile devices, HPC, and data centers where the total cost of ownership is forecast to be dominated by power and cooling costs rather than software or hardware [32]. One way large systems builders have tried to increase computation without increasing power is to employ parallelism. As an example, the NVIDIA GTX480 GPU [4] achieves 1344 GFLOPS[1] single precision at 250

[1] This number is derived from NVIDIA product sheets that state the processor has a 1.4 GHz processor clock and 480 shaders. We assume 2 FLOPs per cycle with FMAC.

watts (max.) using 3 billion transistors while the latest six-core i7 CPU from Intel [3] achieves 518.4 GFLOPS[2] at 130 watts (max.) using 1.17 billion transistors. The Roadrunner supercomputer [33], which incorporates both AMD CPUs and Cell processors, is an example of accelerators being used to increase FLOPS per watt. The trend of increasing parallelism to reduce energy consumption is also becoming more prevalent in the mobile world, where systems-on-a-chip include accelerators to reduce power [34] and multicore applications processors are now available [35].

While parallelism offers the promise of increased performance, reduced design complexity, and a better power-performance trade-off, there are many factors confounding the greater adoption and exploitation of parallel architectures. Increasing the programmability of parallel systems is a key concern with many efforts to build tools and train users underway. The programming models are a topic of active research, but are not the focus of this dissertation. We focus on the scalability concerns for current programming practices and parallel architecture; in the next section we discuss them more fully.

2.2 The Limits of CMP Scalability

Scalability is the capacity of a parallel computer system to continue to provide increased performance commensurate with the increase in processing or memory resources provided to that system. The delivered performance can be measured in terms of the time to complete a fixed task, i.e. *latency*, or the number of tasks that can be done in a fixed time, i.e., *throughput*. In this dissertation, we consider performance scalability in terms of throughput.

In discussing scalability, we assume a parallel hardware system is running soft-

[2]This number is derived from Intel's product sheet for the i7-980x at peak frequency. We assume 2 FLOPs per cycle with fused FMAC as is done with the quoted GPU numbers.

ware in parallel as prerequisite to discussing scalability of the system as a whole. Sequential code is inherently unscalable and is not considered here. However, we do not place constraints on what kind of parallelism the application expresses. In other words, for a scalable system running scalable software, we expect that doubling the number of cores in a single processor doubles the aggregate application throughput. An example is a web server able to process 100 requests per second with 10 ms latency with one CPU that is able to process 200 requests per second with two CPUs while still achieving 10 ms average latency.

Scalability is of great concern in parallel architecture, as it is the means by which one achieves greater levels of performance by adding more cores to a processor, adding more processor sockets on a motherboard, or adding more racks to a datacenter. Scalability can be limited by hardware or software factors. In the era of multicore processors, there is no incentive to upgrade hardware if the application and the hardware are not scalable, since most of the benefit of switching to a newer processor today is the additional cores or added memory bandwidth and is generally not a reduction in latency.

In this section we discuss factors that contribute to limiting the scalability of future CMPs. Most of this dissertation is focused on the hardware factors related to scalability of memory models and cache hierarchies for future CMPs. As such, this section focuses primarily on those topics, and we only introduce software-related topics and more general hardware implications of scaling.

Achieving scalability requires that an application express sufficient parallelism relative to the amount of sequential work present in the program. The sequential work is the critical path that sets a floor on the improved runtime any parallelization strategy could hope to achieve. The relationship between sequential work and parallel scalability, or Amdahl's law [36], is attributed to Gene Amdahl who pointed out the inability of multiprocessors to scale when there is a high degree

of irregularity or data management overhead present in the nonparallel version of the software.

It is necessary also to consider the scalability relative to dataset size. The relationship between dataset size and scalability is known as Gustafson's law [37]. Accounting for larger dataset sizes may provide maintained scalability of an application over multiple generations of CMP. The reason for the continued scalability is that over time users may not want to do the same thing faster, but they may want to do more in the same amount of time. However, even if core counts continue to scale, Gustafson's law may be stymied by the inability of cache, networking resources, and/or memory bandwidth to scale commensurate with the number of cores.

While these simple models of scalability provide insight into how aspects of a parallel application may impede or extend scalability, they fail to account for the full space of design trade-offs [38]. One such trade-off is between the relative benefit of increased per-core area, power, and complexity to achieve faster sequential performance versus the greater degree of compute resources possible with smaller, more area efficient cores [39]. We revisit this trade-off in greater detail throughout the dissertation.

Even when there is sufficient parallelism, scalable applications must limit communication among parallel threads or overlap it with computation, thus hiding the latency. An application that has sufficient parallelism and adopts practices that hide or remove communication latency may still be limited by the parallel hardware upon which it runs. In a cached shared memory architecture, such as the CMPs that we evaluate in this work, the location of each block of data is tracked implicitly by software or explicitly by a hardware mechanism, i.e., cache coherence. When communication between two threads occurs at the software level, it may have the unintended side effect of generating multiple network mes-

sages or increasing the storage necessary for tracking shared state maintained by hardware. The storage and network costs associated with communication among parallel threads of execution is fundamental to the scalability of a CMP. Network communication and coherence storage costs add overhead that may mitigate any performance gained by increased parallelism; the reduction of both is the topic of this dissertation.

As we will show, current approaches for increasing the scalability of CMPs are insufficient at a high core count. A common addition to contemporary CMPs are *probe filters*. As shown in [40, 41], probe filters become a limiting factor for scalability even at a small number of cores. We survey contemporary approaches to scalable cache coherence in the next section. System-level factors can also contribute to limited scalability. A survey of common desktop and workstation applications [13] shows that even after parallelization, these applications fail to make use of the parallel hardware available to them. The problem facing architects is thus not just how to achieve greater hardware scalability at the architecture level, but how we can provide a scalable hardware abstraction that allows software to effectively utilize the parallel hardware to achieve realized scalability at the application level.

2.3 Elements of Parallel Accelerator Design

This dissertation focuses on developing scalable memory models for compute accelerators. We use the Rigel [42] architecture as our baseline. Before considering the memory model of our proposed accelerator design, we provide a top-down motivation of the design choices for the Rigel programming interface. The design choices include the set of functionality to be supported by the architecture and low-level software of a programmable compute accelerator.

Programmable accelerators span a wide spectrum of possible architectural models. At one end of the spectrum are FPGAs, which can provide high compute density and fine-grained configurability at the cost of a very low-level native application programming interface (API). The gate-level orientation of the FPGA interface, i.e., netlist, creates a large semantic gap between traditional programming languages, such as C or Java, and the low-level programming interface (LPI). The semantic gap requires that the programmer make algorithmic transformations to facilitate mapping or bear the loss of efficiency in the translation—and often, both. The other end of the spectrum is represented by hardware accelerators and off-load engines tightly coupled to general-purpose processors. Examples include TCP/IP and video codec accelerators incorporated into systems-on-a-chip (SoC). Here the LPI is an extended version of the traditional CPU LPI, i.e., the ISA, and thus makes an easier target for programmers and programming tools.

Akin to an instruction set architecture, the LPI is the interface between the applications development environment and the underlying software/hardware system of the accelerator. We show the entire software stack for accelerator architectures in Figure 2.1. The LPI subsumes the ISA: as with any uniprocessor interface, the accelerator interface needs to provide a suitable abstraction for memory, operations, and data types. Given that programmable accelerators provide their performance through large-scale parallel execution, the LPI also needs to include primitive operations for expressing and managing parallelism. The accelerator LPI needs to be implemented in a scalable and efficient manner using a combination of hardware and low-level system software.

What is desirable from a software development point of view is a programmable accelerator with an LPI that is a relatively small departure from a conventional programming interface. The LPI should also provide an effective way to exploit the accelerator's compute throughput. In this section, we motivate the trade-offs

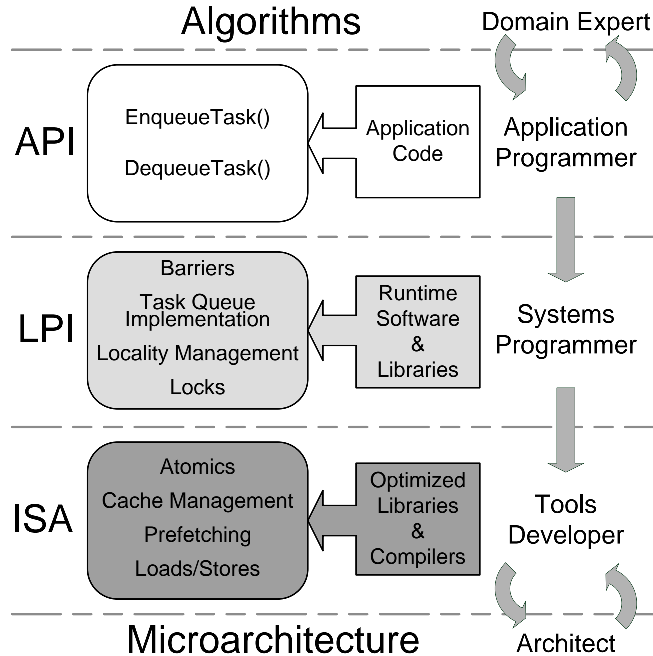


Figure 2.1: Software stack for accelerator architectures. API: application-level programming interface, LPI: low-level programming interface, ISA: instruction set architecture.

made in Rigel between generality in the LPI and accelerator performance. To that end, we describe the *elements* that we identify as necessary for supporting these objectives. The elements described include the execution model, the memory model, work distribution, synchronization, and locality management.

2.3.1 Element 1: Execution Model

The execution model is the mapping of the task to be performed, specified by the application binary, to the functional units of the processor. The choice of execution model is ultimately driven by characteristics of the application domain and its development environment. The overarching goal for accelerators is for the execution model to be powerful enough to efficiently support common concurrency patterns, yet be simple enough for an implementation to achieve high compute density. The execution model encompasses the instruction set, including

its level of abstraction and use of specialized instructions, static versus dynamic instruction-level parallelism, e.g., VLIW versus out-of-order execution, and SIMD execution versus MIMD.

The execution model may be virtual, thus decoupling the software instruction set from the hardware instruction set. Doing so allows for just-in-time optimization as is done in Java compilers where the binary targets the Java Virtual Machine [43] and in the LLVM compiler [44] where an intermediate representation of the code, the virtualized execution model, can be optimized independently of the underlying hardware execution model. However, while such systems allow for binary portability and backwards compatibility, it is unclear whether performance portability can be assured when moving from one parallel hardware platform to another.

The goal for Rigel is to develop a general-purpose execution model suitable for compact silicon implementation. The choice of the SPMD execution model is backed by previous studies and experience that show that the SIMD model imposes undue optimization costs for many irregular applications. Mahesri et al. [12] show that even considering the area benefit of SIMD, some parallel applications scale poorly on long vector architectures, reducing the *effective* compute density of the accelerator. In fact, it has been shown that code written for a SIMD architecture can match or exceed the performance on a MIMD system [45]. However, with proper hardware and software support, the cost of mapping of MIMD-friendly codes to SIMD can be mitigated [46, 47].

2.3.2 Element 2: Work Distribution

When an application reaches a section of code suitable for parallel acceleration, work is systematically distributed to available chip resources, ideally in a fashion

that maximizes the throughput of the accelerator. With Rigel, we adopt a task-based work distribution model where parallel regions are divided into parallel tasks by the programmer, and the underlying LPI provides mechanisms for distributing tasks across the parallel resources at runtime in a fashion that minimizes overhead. Such an approach is more amenable to dynamic and irregular parallelism than approaches that are fixed to parallel loop iterations.

In Chapter 3 we discuss the actual programmer interface for the Rigel Task Model (RTM), an API for enqueing and dequeuing tasks, supported by a small number of primitives in the underlying LPI. We show that RTM can support fine-grain tasks at negligible overhead at the scale of 1000 cores.

2.3.3 Element 3: Synchronization

Selection and implementation of synchronization primitives abounds in the literature. Blelloch [48] describes the generality of reduction-based computations. The implementation of barriers in particular has been accomplished with cache coherence mechanisms [49], explicit hardware support such as the Cray T3E [50], and, more recently, a combination of the two on-chip multiprocessors [51]. Using message passing networks to accelerate interprocess communication and synchronization was evaluated on the CM-5 [52]. Interprocessor communication using in-network combining in shared-memory machines such as in the NYU Ultracomputer [53] and using fetch-and-ϕ operations as found in the Illinois CEDAR computer [54] have also been studied. These designs give relevant examples that influence our work as we reevaluate the trade-offs of past designs in the context of single-chip, thosuand-core, hierarchical accelerators.

The ability to support fine-grained tasks, and thus a high degree of parallelism, requires low-latency global synchronization mechanisms. Limiting the scope to

data- and task-parallel computation focuses the support required for Rigel to two classes of global synchronization: global barrier support, which is required to synchronize at the end of a parallel section, and atomic primitive support, which is useful for supporting shared state, such as updating a global histogram using the atomic increment primitive. Local atomic operations, which allow read-modify-write operations to occur at higher levels of the cache, are invaluable for achieving scalable performance when designing hierarchical algorithms and task management systems. Both cases are discussed in later chapters.

2.3.4 Element 4: Locality Management

Locality management involves the co-location of tasks onto processing resources with the goal of increased local data sharing to reduce the latency and frequency of communication and synchronization among co-located tasks. Locality management can be performed by a combination of programmer effort, compiler tools, runtime systems, and hardware support. In programming parallel systems, performing locality-based optimization constitutes a significant portion of the application tuning process. An example of locality management is blocked dense matrix multiply, in which blocking factors for parallel iterations increase the utility of shared caches by maximizing data reuse and implicit prefetching across threads while amortizing the cost of cache misses.

Accelerator hardware and programming models also rely heavily on locality management. Modern GPUs such as the NVIDIA G80 make use of programmer-managed local caches and provide implicit barrier semantics at the warp-level using SIMD execution [4]. The CUDA programming model allows for the programmer to exploit the benefits of shared data using the shared memories of the GPU, fast synchronization across warps using `__syncthreads` primitives, and the

implicit gang scheduling of threads through warps and thread blocks. Models such as Sequoia [55] and HTA [56] demonstrate examples of how to manage locality on accelerators such as the Cell Processor [57] and for clusters of workstations.

Memory bandwidth has historically lagged available compute throughput; thus, the memory bandwidth a single chip can support limits achievable performance [58]. The cost of communicating has grown to hundreds of cycles to perform cross-chip synchronization or memory operation between two cores [59]. Because they are optimized for compute throughput on kernels, accelerators tend to have smaller amounts of on-chip cache per core. The fraction of per-core cache allocated to each processing element in modern accelerators, which can be on the order of kilobytes [4], is a fraction of the megabytes per core available on a contemporary multicore CPU. The communication latency, synchronization overheads, and limited per-core caching all indicate that the locality management interface is a critical component of an LPI.

2.3.5 Element 5: Memory Model

The design of a memory model for a parallel programmable system involves a choice of memory hierarchy, including software-managed memories such as those found in the Cell Processor or multiple specialized address spaces found in GPUs [4], as well as choices regarding explicit versus implicit interprocessor communication and allowable memory orderings. Trade-offs between these choices are hard to quantify, but it is understood that one can generally reduce hardware complexity, thus increasing compute throughput, by choosing simpler, software-controlled mechanisms, albeit at additional complexity in software development.

The baseline model of computation for uniprocessor machines is that of von Neumann where a single stream of computation fetches data from a single memory.

In this model there is no intermediate level of storage between memory and the instruction stream, there is no remapping of addresses, and there is no sense of ordering between memory operations other than program order. The simple model, while still serving as a mental model employed by programmers today, has long since been replaced in hardware by processors with multiple interconnected cores and multiple levels of shared or private caches.

The conventional cache hierarchy of a uniprocessor is meant to exploit the common patterns of temporal and spatial locality found in most workloads to provide lower latency for memory accesses, which translates into increased sequential performance. Caches are functionally transparent to software, thus reducing programmer investment in transcribing algorithms from the von Neumann mental model into code. Hardware-managed caches are an alternative approach in lieu of scratchpad memories where the illusion of a single memory is broken by the need to explicitly move data between levels of the hierarchy.

Processors with multiple cores add further constraints to the design of the cache and interconnect. Both memory ordering and coherence guaranties between cores come from the need to support a reasonable machine model for software to target. Common patterns from software demonstrate the need to reduce contention at higher levels of the hierarchy. Physical limitations of the design constrain port count, which determines the number of concurrent accessors permissible at a particular cache bank in one cycle, while signaling delays limit the size of arrays at various levels in the cache. Interconnecting a small number of cores can be accomplished with an area-efficient shared structure such as a bus or a more complex, concurrent interconnect such as a crossbar.

Accelerators represent an evolution of the trends motivating cache design. The large numbers of cores represent a challenge for hardware designers who must balance programmability against physical implementation constraints. Software

developers too face challenges, as they must reason about ordering, contention, and—no matter how transparent functional—the impact on performance of the caching hierarchy and interconnect topology. As the number of cores grows, the relative size of per-core structures shrinks and can lead to area-inefficient memory arrays and replication of possibly complex controller logic.

The LPI needs to incorporate a set of rules that defines allowable orderings for access to shared state. Moreover, the interface also must allow the user to manage data accesses in such a way that it provides reasonable orderings when proper synchronization is used. Examples here are global versus local cache accesses, memory barriers, flush instructions, etc. This critical piece of the LPI, the memory model for accelerator architectures, is the key contribution of this dissertation.

2.3.6 Low-Level Programming Interface

Now that we have introduced the general concept of an LPI and discussed its components, we conclude this section with an overview of the complete Rigel LPI, addressing the points raised in the earlier subsections. The low-level programming interface to Rigel supports a simple API for packaging up tasks that are managed using a work queue model. The individual tasks are generated by the programmer, who uses the SPMD execution model and single global address space memory model in specifying the tasks. It is the responsibility of the work distribution mechanism, the RTM implementation, to collect, schedule, and orchestrate the execution of these tasks. Execution of these tasks is based on the prevalent bulk synchronous parallel (BSP) [60] execution model, which is also the de facto model for many other accelerator platforms such as CUDA-based GPUs. With BSP, a parallel section of tasks is followed by barrier synchronization, followed by the next parallel section.

The Rigel LPI supports task queues as a means to distribute tasks. Global synchronization is provided by an implicit barrier when all tasks for a given phase of the computation have completed, forming an intuitive model for developers. The Rigel LPI also provides a means to implicitly (at barriers) or explicitly (under software control) make updates to globally visible shared state before entering a barrier to provide a coherent view of memory to programmers.

Locality management at the low-level programming interface is provided via a combination of mechanisms to co-locate groups of tasks to clusters of cores on chip and to manage the cache hierarchy. Much of the locality management is provided implicitly by hardware-managed caches that exploit temporal and spatial locality, as with a typical CPU. A programmer can tune the effectiveness of these implicit structures through co-location of tasks to increase reuse of shared data. To that end, the Rigel LPI supports grouping of tasks that have similar data access streams, thus increasing the effectiveness of local caches for co-located tasks. Similarly, tasks that require local synchronization can be co-located onto the same cluster of cores, thus synchronizing through the local caches with less overhead than with global synchronization. To provide explicit control when necessary, the Rigel LPI supports cache management instructions, explicit software-controlled flushes, memory operation that bypass local caches, and prefetch instructions for explicit control for performance-minded programmers to extract higher performance from the accelerator when desired.

With the LPI for Rigel, we choose to present application software a general-purpose memory model typical of multicore CPUs: a single global address space across the various cores of the accelerator. The address space can be cached and is presented to the programmer in a coherent way; however, the actual hardware may not provide coherence directly. With such a model, managing the memory hierarchy can be done implicitly by the software. Interprocessor communication

Table 2.1: Design parameters for coherence on throughput-oriented architectures similar to our baseline. We use the following abbreviations with values used in our evaluation in parentheses: **n**: Potential sharers (128). **S**: Number of active sharers ($0 \leq \mathbf{S} \leq 128$). **i**: Number of sharer pointers in limited schemes (4). **C**: Sharers covered by a coarse-grained vector bit (often 4). $L2_s$: Number of sets per sharer L2 (128). $L2_w$: Number of ways per sharer L2 (16). **M**: Total lines in memory (roughly 128 million for 4GB of memory). **E**: Maximum number of cache lines tracked by on-die structure ($1 \leq \mathbf{E} \leq L2_s \times L2_w \times n$). **F**: Probe filter size. **T**: Tag size.

		Snoop [61]	Snoop+Filter [62, 63]	Duplicate Tags [64]
Storage		$\epsilon L2_s L2_w \times \mathbf{n}$	$(\epsilon L2_s L2_w + \text{F}) \times \mathbf{n}$	$\mathbf{T} \times L2_s L2_w \times \mathbf{n}$
Broadcast Frequency		Always	Filter Misses	Never
Network Traffic		High	Low for small **n**	Low
On-die	Structure	Bits w/L2 Tag	Tagged SRAM	CAMs
	Location	Local		Centralized (@LLC/DRAM)
	Capacity	Negligible	O(10-100kB)	O(1MB)
	Associativity	NA	Low [62]/NA [63]	$L2_w * \mathbf{n}$

			Sparse [65, 66]		
		Full Directory [67]	Full-map [68]	Coarse-vector	Limited
Storage		$\mathbf{n} \times \mathbf{M}$	$\mathbf{n} \times \mathbf{E}$	$\frac{\mathbf{n}}{\mathbf{C}} \times \mathbf{E}$	$\mathbf{i} \times \log_2 \mathbf{n} \times \mathbf{E}$
Broadcast Frequency		Never		Multicast of **C**	if ($\mathbf{S} > \mathbf{i}$)
Network Traffic		Low		$\propto \mathbf{C}$	Low if($\mathbf{S} \leq \mathbf{i}$)
On-die	Storage	SRAM/DRAM	Banked/Tagged SRAM		
	Location	Centralized (@LLC/DRAM)	Centralized (Banked)		
	Capacity	O(1 GB)	O(4MB)		O(1MB)
	Associativity	If caches, arbitrary	$1 \leq A \leq (L2_w * \mathbf{n})$		

is implicit through memory, reducing the semantic gap between high-level programming and the LPI. Providing implicit support for the memory model creates an implementation burden on the underlying LPI: if the address space is cached, which is required to conserve memory bandwidth, then one needs to consider the overheads of caching and also coherence, discussed with respect to hardware and software in Chapter 3 and Chapter 5, respectively.

2.4 Design Space of Cache Coherence Protocols

In this section, we present a range of design choices for implementing coherence on a CMP. The taxonomy of choices is given in Figure 2.2. We discuss messaging overhead and area required for coherence state tracking, which are the first-order concerns for building a scalable on-die coherence architecture. We conclude by discussing the design space of sparse, limited directories, which is the subclass of directory protocols used throughout most of this dissertation.

We limit our evaluation to directory schemes due to the excessive bandwidth

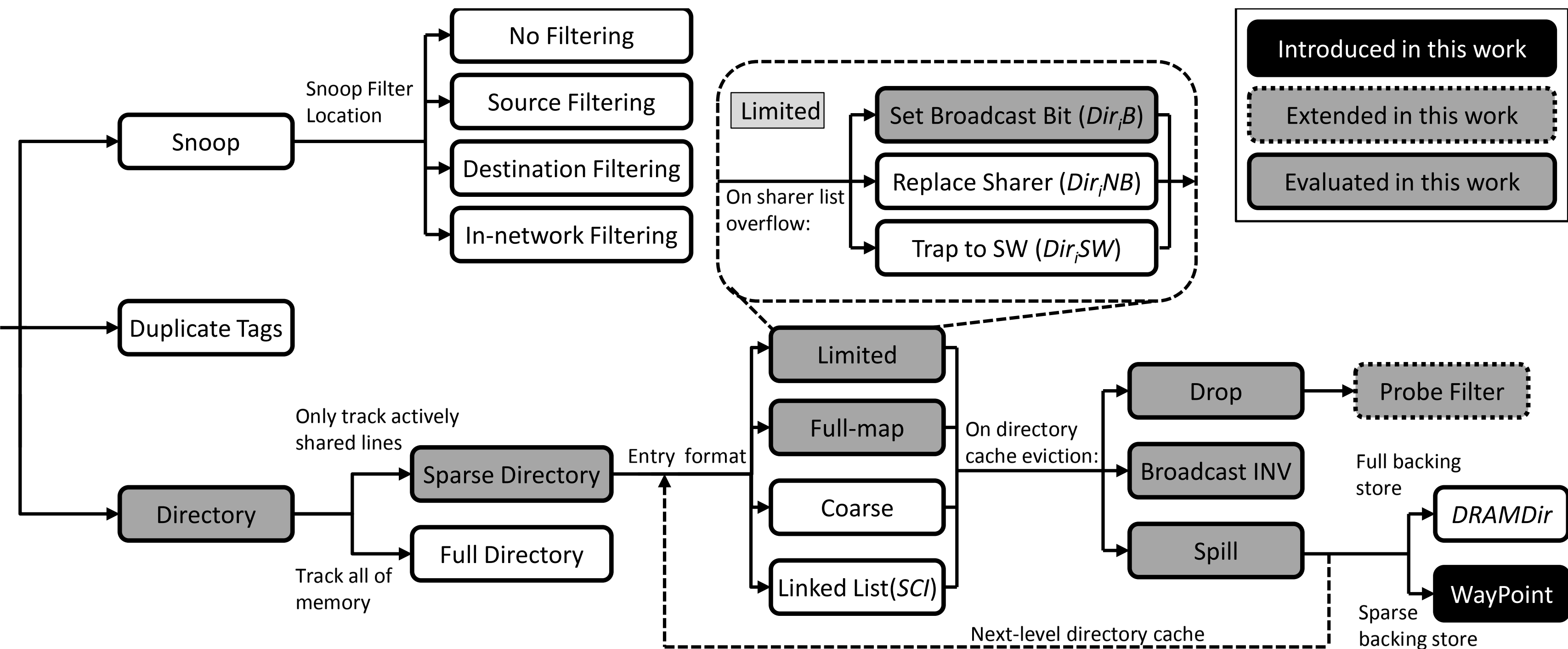


Figure 2.2: Classification of cache coherence schemes. We evaluate sparse directories with full-map (Dir_nNB) and limited (Dir_4B) directory entries. We evaluate probe filtering, broadcast invalidate, and WayPoint directory cache eviction policies. (Figure courtesy of Matt R. Johnson.)

requirements of snoop-based protocols [61] and the highly associative lookups required by duplicate tag schemes [64]. The high-level trade-offs between these approaches are summarized in Table 2.1. Within the class of directory schemes, we consider *full directories* and *sparse directories*. A full directory has one entry per cache line in the address space. The state for a full directory may be held strictly in off-chip memory [67] or cached on-die [68]. Even assuming a compact directory entry format, a full directory requires hundreds of megabytes to gigabytes of storage for systems with gigabytes of memory and hundreds of sharers. In contrast, sparse directories [65] only track lines that are actively shared. For a CMP, the storage overhead of a sparse directory is proportional to on-die cache capacity rather than the size of the address space. The pathological case for sparse directories is when every line in every sharer cache is unique. We refer to sparse directories with enough entries to handle this case without evictions as *complete sparse directories*, and to those with fewer entries as *subset sparse directories*. We evaluate several variants of complete and subset sparse directories throughout this dissertation.

Full directories may be cached [69], but entries evicted from the directory cache must be written back to memory. Sparse directories may also need to evict valid entries. One choice for handling evictions is to have the directory invalidate all sharers before evicting. A directory miss then implies zero sharers, and the directory maintains perfect sharing information. A second approach rebuilds the sharing state when a miss occurs at the directory. This approach is referred to as *probe filtering*, because the directory caches can be viewed as caching some number of sharing vectors to eliminate broadcast probes for those lines. On a miss, the directory can either broadcast invalidate to ensure there are no sharers, or broadcast probe to reconstruct the sharer list. Modified lines are dealt with using a separate writeback message when a probe discovers the line in a modified

state or by never evicting modified entries from the directory. We investigate probe filtering further in Chapter 6. A third approach is to spill the entry to a second-level directory cache or a backing store in memory. If evicted directory entries are spilled to memory, the backing store can be full [68] or sparse. In Chapter 7, we propose a complete sparse directory cache scheme called WAYPOINT which includes a sparse backing store in cacheable system memory.

Sparse directories can be further classified by their entry format and their eviction policy. Consider a system with n potential sharers. Full-map entries (Dir_nNB) maintain a complete list of sharers and thus support all sharing patterns equally well, but require n bits per entry. Empirically, we have found that our workloads have bimodal sharing patterns where each line has either a small number of concurrent sharers or close to n concurrent sharers. Such sharing patterns are efficiently supported by *limited directory* entries which maintain a list of i pointers to sharers and only require $i \log_2 n$ bits per entry. The benefit of this approach is that for the two common cases, where $s \approx n$ and $s \leq i$, there are few additional messages sent in the former case and none sent in the latter compared to implementing a much more costly full-map directory. Limited directory schemes differ in how they respond to sharer list overflows; the entry can either revert to broadcasting on the next coherence state transition, invalidate an existing sharer to replace it with the requester that caused the overflow, or trap to a software routine which can dynamically decide between the two or maintain a sharer list in memory [70, 71].

While a variety of directory schemes were originally developed for use in systems where aggregate memory bandwidth and capacity scaled in tandem with processor count, none were engineered specifically for CMPs, which represent a different design point. Moreover, none of these designs considered heterogeneous systems where highly parallel compute accelerators and general-purpose cores are

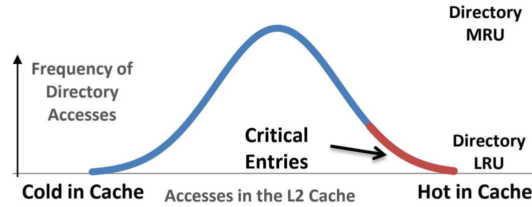


Figure 2.3: Illustration of the difficulty in selecting a directory replacement policy due to the information disconnect between L2 caches and the directory.

integrated on a single device. While last-level cache capacity and total available memory may scale with the number of cores, the memory bandwidth and local cache capacities are not scaling commensurately with the number of cores. Moreover, the latency to memory has grown considerably. Furthermore, when coherence state was stored off-die in memory, die area dedicated to compute did not need to be compromised for the sake of coherence storage. However in a CMP with coherence state tracked on-die, the trade-off is present, thus adding an extra dimension over which to optimize. These factors lead us to focus on directory storage costs and network messages in our evaluation of CMP cache coherence architectures.

For implementations that support incomplete directory caches, directory entries containing active sharers may need to be evicted to accommodate newly arriving requests. Directory cache replacement policies must deal with the disconnect between directory access recency and core access recency, as shown in Figure 2.3. The most frequently accessed entries are those frequently requested and released by the L2s. An MRU policy would evict these entries, possibly forcing actively shared data to be invalidated. On the other hand, an LRU policy is able to remove cold data, but runs the risk of invalidating lines which are hot in the L2 and unlikely to generate directory traffic. Moreover, throughput-oriented workloads typically contain a large amount of touch-once and private data which may thrash the directory cache and evict widely-shared data.

2.5 Discussion and Summary

In this section we provide an overview of the challenges facing CMP scaling. We focus on the hardware factors that will limit scalability of future systems if these factors are not addressed. With the proliferation of parallel architectures, increased heterogeneity in the hands of developers, and the re-purposing of accelerators for general-purpose computation, we see an opportunity and a need to raise the level of abstraction that developers target. We enumerate the design goals of a raised abstraction and use these elements to guide our investigation of highly parallel accelerator architectures and their memory models. To better understand what techniques are available today for providing a coherent view of memory to the developer, we provide a taxonomy of proposed techniques for scalable cache coherence. With an understanding of where parallel development is going, the challenges faced by the systems builders attempting to scale out CMPs toward 1000-cores, the design goals of a 1000-core processor, and the state of the art in coherence protocols, we are prepared to investigate the characteristics of the workloads these systems will target and novel approaches to scaling the memory models for these workloads.

CHAPTER 3

Architecture Baseline

In this chapter we review our baseline architecture and programming model. The proposed work builds upon the platform described in this chapter. For those readers familiar with Kelm et al. [42], this section may be skipped.

3.1 Rigel Architecture

We use the Rigel architecture [42] as the baseline for our evaluation. Rigel is a 1024-core MIMD compute accelerator that targets highly task- and data-parallel applications in the areas of computer vision, imaging, and physical simulation that scale up to thousands of concurrent tasks, collectively known as visual computing workloads. The design goal of Rigel is to provide high compute density by minimizing per-core area while still enabling a conventional programming model. Density is improved by removing features found in conventional designs that are of minimal benefit to the workloads targeted by Rigel. A block diagram of the baseline Rigel architecture is given in Figure 3.1.

3.1.1 Overview

The fundamental processing element of Rigel is an area-optimized 32-bit dual-issue in-order core. Each core can issue up to two ALU operations per cycle, one of which may be a branch. The pipeline has one single-precision floating-point unit that supports floating-point addition, multiplication, reciprocal square root,

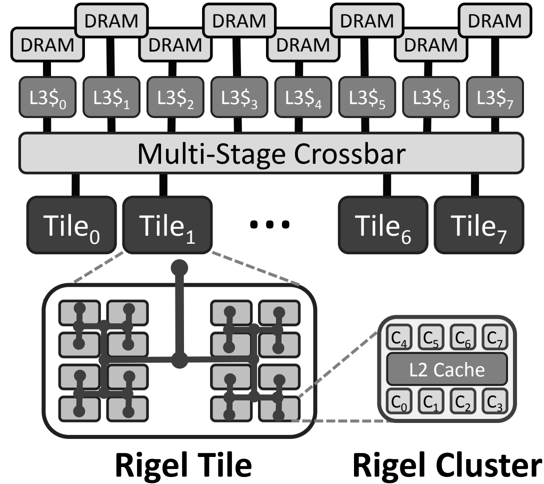


Figure 3.1: Baseline Rigel Architecture.

and division. Integer division is implemented in software. The single instruction stream for each core is supported by an independent fetch unit which executes a RISC instruction set. The choice of the MIMD execution model and simple two-issue in-order cores is supported by recent studies [12, 39] that find such a configuration to be optimal across a number of parallel workloads targeting a throughput-oriented architecture such as Rigel.

Eight Rigel cores are attached to a unified L2 cache named the *cluster cache*. The cores, core-to-cluster-cache interconnect, and the cluster-to-global interconnect logic comprise a single Rigel *cluster*. The core-to-cluster-cache interconnect is a pipelined split-transaction bus with two cache-line-wide lanes. Each core has independent L1 data and instruction caches. The core-level caches are kept coherent by snooping the core-to-cluster interconnect. The L1 data caches are write-through to the L2. In effect, the cluster acts logically as an eight-way SMP. Each cluster cache is 64 kB in size. In aggregate, the 128 Rigel clusters provide 8 MB of L2 cache.

Clusters are connected and grouped logically into a *tile* using a bi-directional tree-structured interconnect. Eight tiles are distributed across the chip. The

tiles aggregate traffic from the clusters and attach to the global L3 cache banks via a crossbar interconnect. The L3 cache is configured as a shared, unified, centralized, last-level cache. The L3 caches provide buffering for multiple high-bandwidth memory controllers. Global L3 cache banks provide a serialization point for inter-cluster shared data for maintaining a coherent view of memory. No communication occurs between clusters independent of the L3 cache. There are 32 L3 cache banks for the entire chip, arranged into 8 clusters that share ports on to the global interconnect, totaling 4 MB of last-level cache.

Our design incorporates 8 GDDR5 memory controllers. The memory controllers use a banked first-ready, first-come, first-serve (FR-FCFS) [72] policy with 16 scheduler entries per DRAM bank. We use open page row switching. To exploit DRAM row locality to increase memory bandwidth, L3 misses trigger a block prefetch of four lines.

3.1.2 Cache Management

The material in this section pertains to the baseline Rigel system supporting software-managed coherence. Discussion of the modifications necessary to support hardware coherence and hybrid coherence is delegated to Chapter 7 and Chapter 8, respectively. Briefly, the hardware coherence extensions to the baseline presented in this chapter include extra messages for L2 cache evictions of clean data, request messages on write misses at the L2, probe requests and replies, and hardware structures co-located with the last-level cache for tracking coherence state, i.e., a coherence directory.

All cores share a single global address space. Cores within a cluster have the same view of memory due to the shared cluster cache, while global coherence between clusters is not maintained by the hardware. When serialization of accesses

between clusters is necessary, the global cache is the point of coherence. To access each level of cache directly, Rigel implements two classes of memory operations: *local* and *global*.

Local memory operations are intended to constitute the majority of memory operations. Low-latency and high-bandwidth memory accesses are achieved using local operations. Local read operations are cacheable at the cluster cache, but are not kept coherent between clusters by hardware. Local memory writes follow a writeback policy at the cluster cache; on eviction from the cluster cache, modified data is written back to the global cache. From the perspective of the programming model, local operations are used for accessing read-only data, private data, and data that is shared intra-cluster. Rigel provides a cluster-level load-linked/store-conditional pair for atomic operations at the cluster cache. The period between two global barrier operations is referred to as an *interval*. Software must enforce cache consistency when inter-cluster read-write sharing exists within an interval. On the other hand, read-to-read sharing is permissible without additional software management.

Global loads, global stores, and atomic read-modify-write operations on Rigel bypass the cluster cache and complete at the global cache, which serves as the point of global coherence. Global memory operations, in essence, enforce a write-through semantic at the L2 for the data upon which they operate. Memory locations operated on solely by global memory operations are kept coherent across the chip. Globally visible operations are key to supporting system resource management and synchronization for a chip that supports cache coherence in software.

Global memory operations also enable fine-grained inter-cluster communication by way of the global caches without the need to obtain ownership as is necessary in invalidation-based coherence protocols. To be clear, what is saved in the Rigel model is the *exposed* latency of cluster-to-cluster communications. On

a conventional CMP with cache coherence, every cache-to-cache transfer requires a network traversal that is exposed to all pending accessors. The Rigel model allows those accesses to be pipelined and only exposes the latency of performing the read-modify-write operation at the L3. Even so, the cost of global memory operations is high relative to local operations due to the greater latency of accessing the global caches versus the local cluster caches. Furthermore, the achievable global memory operation throughput is limited by the number of global cache ports, the latency of performing a global operation, and cluster-to-global cache interconnect bandwidth.

The cache controller and coherence policies are designed to tolerate an unordered network. Any message from any cluster can be reordered between the L2 and the L3 by the network. Messages sent by any one L2 are not guaranteed to arrive at the L3 in the same order as they were sent. Similarly, replies from the L3 are unordered. As such, the L2 cache controller handles global operations with special care to avoid ordering violations as dictated by the memory model. L2 fill operations resulting from local cache accesses can be merged at any point. When the response returns from the network, the L2 fills the line and marks all words valid. Requesters then access the line as if the line was always cached. Note that for our cache coherence additions to the baseline Rigel design, we force requests to complete in-order at the L2 to avoid starvation.

If a global operation is issued to a line that is cached in the L2, that line must be invalidated or, if modified, written back to the L3 before the global operation can be issued to the network. If a core issues a global operation request and there is an L2 fill pending request present at the L2 cache controller, the cache controller waits for the fill to complete and then performs the invalidation and writeback operations described above before issuing the global operation request to the network. Local operations that miss in the L2 when a global operation is

pending generate a fill request, but the fill request is not issued to the network until the global operation completes. Only one global operation per address may be outstanding at the L2 cache controller. To provide proper memory consistency across cores, only one global operation per core may be outstanding.

The software protocol we build on top of Rigel's cache hierarchy, the Task-Centric Memory Model that is described in greater detail in Chapter 5, works at the word granularity. Word-granularity in the coherence protocol is advantageous because the memory operations have the granularity of a word in the Rigel architecture. However, caching occurs at the line granularity, nominally 32 bytes in our design. The difference in granularity requires further consideration to avoid lost updates to shared cache lines under our software protocol. If the coherence granularity were a full cache line, a writes by multiple cores to the same line would result in a race condition even if the write sets within the word did not overlap. For a hardware cache coherent design, the discrepancy between sharing granularity and memory operation granularity would not present a correctness problem; however, such sharing could manifest as *false sharing*. False sharing is the situation where two cores concurrently access distinct sets of words that reside on the same line. False sharing would only affect performance but not correctness. The mutual exclusion property for writes under hardware cache coherence would disallow two updates to distinct words on the same line to race.

To lift the burden of software management of false sharing in Rigel, we include per-word dirty and valid bits in the L2 caches. When a core writes to a line, the L2 allocates an entry for the line, if not already present, and sets the dirty bit for the written word. The dirty and valid bits are propagated to the L3 where the valid and dirty words are merged with the existing entry, when present, or a new dirty entry is allocated with only the dirty and valid words set as valid at the L3. For writes to distinct words in the same line, the merging at the L3 ensures

that no updates are lost, since the words being updated are mutually exclusive. If true sharing exists, it represents a race in the software protocol. This race is no different than one that would occur in a hardware coherent system with a relaxed memory model. The hardware at the L3 handles races by allowing the last update to persist; the hardware is unaware that a race exists. While allowing races may seem to be an additional burden brought on by the software-managed coherence protocol, it is no different from unsynchronized writes in a cache coherent system.

3.1.3 Coherence and Synchronization

Without hardware mechanisms to enforce coherence between cluster caches, alternative approaches must be used for conflicting shared data access. Write-shared data on Rigel could be kept coherent between sharers by forcing all modification to be made using global stores and all reads using global loads; however, the cost of using only global memory operations would be high and strain global network and cache resources. From a software perspective, global stores can be thought of as `put` operations in a partitioned global address space (PGAS) model [73–76]. Similarly, a global load can be thought of as a `get`.

One of the key motivations for RTM, our task-based programming model, which is described in greater detail in Chapter 5, is the low frequency of inter-core write-shared data *between* two consecutive barriers. Instead, most read-write sharing occurs *across* barriers and RTM exploits this fact to hide or avoid long-latency global memory operations by implementing a software-managed coherence protocol on top of the Rigel architecture.

The sharing pattern present in our target workloads allows applications targeting RTM to leverage local caches for storing shared data between barriers and then lazily to make modifications globally visible. Lazy updates can be performed

as long as coherence actions performed to output data are complete before a barrier is passed. Rigel enables software management of cache coherence by RTM in two ways. One is by providing instructions for explicit cluster cache management that include cache flushes and invalidate operations at the granularity of both the line and the entire cache. Explicit cluster cache flushes update the value at the global cache, but do not modify nor invalidate copies that may be cached by other clusters.

The second is broadcast invalidation and broadcast update operations that are provided to allow software to implement data synchronization and wakeup operations that rely on invalidation or update-based coherence in conventional cache coherent CMP designs. Broadcast operations will be revisited in the context of parallel reductions and synchronization barriers.

Note that the baseline Rigel implementation does not eliminate cache coherence from hardware, but instead minimizes the hardware cost of maintaining a coherent view of memory by adopting a combined software-hardware approach. The hardware mechanisms for maintaining coherence in Rigel, such as global operations that stall and complete at a high level of the cache hierarchy, trade off complexity and generality for implementation efficiency. Rigel does provide loci of coherence at the cluster, but without the need for additional coherence state or logic due to our choice of implementation.

3.1.4 Scalability

In Kelm et al. [42] we evaluated the scalability of the Rigel architecture in simulation using a set of data- and task-parallel workloads. Expanded and updated scalability results are shown in Figure 3.2. The figure demonstrates that we can achieve near-linear scalability for many of our benchmarks and within 3× of linear

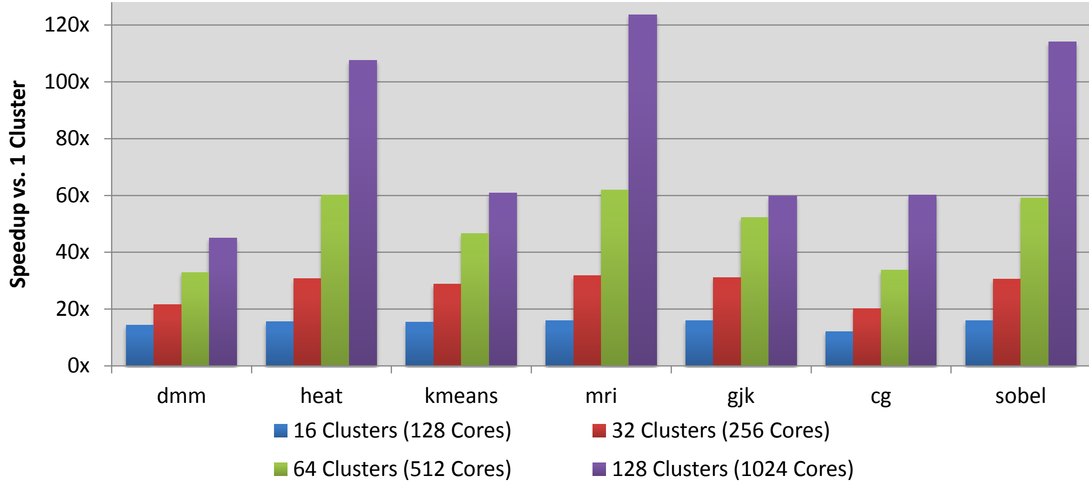


Figure 3.2: Baseline kernel speedup relative to a single eight-core cluster.

speedup across all workloads at 1024 cores. A more thorough analysis of the 1024-core configuration and our extensions to the baseline architecture are presented in Chapter 9.

3.2 Rigel Task Model

The Rigel Task Model is a queue-based low-level programming model, described in [42, 77] that enforces coherence in software and performs synchronization using barriers. The Rigel ISA provides instruction primitives useful for implementing task management, such as local and global atomic operations, but does not provide explicit support for task management. Moreover, the lack of hardware-managed coherence represents an interesting challenges for the design of a parallel runtime system.

In this section we describe the relevant pieces of the API of RTM to provide background and the implementation details relevant to supporting the Task-Centric Memory Model.

3.2.1 Software API

The software API for RTM is composed of basic operations for managing the resources of queues located in memory and inserting and removing units of work from those queues. Applications are written for RTM using a single-program multiple-data (SPMD) execution model where all cores share a single address space and application binary. The programmer defines tasks that are inserted and removed from queues between barrier operations. The barriers thus provide a partial ordering of tasks. The model fits the bulk-synchronous processing (BSP) pattern first described by Valiant [60].

Barriers are used to synchronize the execution of all cores using the queue and to define a point at which all locally cached non-private data modified during that interval must be made coherent. Coherence is enforced by writing back modified output data to the global L3 cache and invalidating read-only non-private input data in the cluster L2 cache. Write-shared data within an interval must be specified by the programmer and is generally implemented by global memory operations on Rigel. The details of the protocol governing these actions are discussed further in Chapter 5. Intrinsics are provided by the API for global memory operations and atomic operations that are kept coherent across tasks within an interval.

3.2.2 Queue Management

RTM provides the following set of API calls to the programmer: `TQ_Create`, `TQ_EnqueueLoop`, and `TQ_Dequeue`. `TQ_Create` allocates resources for the queue and makes it available to the system. Each `TQ_Dequeue` action operates on a single *task descriptor*. A unique task descriptor is generated for each task enqueued and contains two user-defined word-sized data fields and two parameters set by the

runtime to a range that can represent values such as loop iterations. For tasks that require more than two words for input parameters, the fields can be used for pointers to auxiliary data structures. The `TQ_EnqueueLoop` operation provides a single operation to enqueue a DO-ALL-style parallel loop similar to the loop operation available for Carbon [78]. The runtime uses parameters to the enqueue call to select the proper range to deliver to dequeuing cores.

An initialized queue can be in one of the following states: *tasks-available*, *empty*, or *completed*. A newly-initialized task queue, or more generally any initialized task queue without available tasks but not all cores blocking on dequeue, will be in the *empty* state waiting for tasks to be enqueued. Any core that attempts a dequeue operation with an empty queue will block. When tasks are enqueued, the state of the queue becomes *tasks-available*. When tasks are available, dequeue operations return tasks without blocking. If cores are blocking on the task queue and the queue transitions to the tasks-available state, blocking cores are allocated newly available tasks and become unblocked. Tasks are removed in–order from the front of the queue, but may complete in any order between barriers.

The *completed* state is used to provide an implicit barrier in the Rigel Task Model. When all cores participating in a barrier interval have completed all tasks, all cores will be blocking on the task queue and the task queue will transition into the `completed` state. When the `completed` state is reached, a barrier is executed. Although our task generation operation does not have the synchronization semantics found in other models, such as TBB [79] and Cilk [80] where a fork operation is a synchronization between parent and child, the semantics of the completed state are such that tasks can be enqueued at any point within an interval—tasks are not constrained to only be enqueued at the start of an interval. An example of where enqueing tasks during an interval may be useful is in the traversal of a tree structure where sibling subtrees can be processed in parallel, but the number

of tasks is not known a priori.

3.2.3 Implementation

RTM is a multi-level hierarchical task queuing system running on Rigel. Note that the implementation detailed here allows for a task queuing system to be built without the use of global cache coherence but rather through the use of memory operations that bypass possibly incoherent local caches. Coherence management is intertwined with the RTM implementation. Multiple policies for managing coherence are described and evaluated in Chapter 9.

There is a single circular global task queue comprising groups of pointers to task descriptors. The global task queue is located in the application's address space that is resident in main memory or, more likely, cached at the L3, avoiding coherence concerns. The insertions and deletions at the global task queue are synchronized using head and tail pointers that are updated atomically using global atomic operations. Task descriptors inserted into the global task queue are not linked into the task queue until they are flushed to the L3 by the enqueing core. On enqueue operations, the enqueing core fills in the task descriptor locally, flushes it to the L3, and then inserts it into the global task queue. Reducing the exposed latency of enqueue is critical for achieving high speed fanout of work. Here the only exposed latency is the read-modify-write operation to link in the new descriptors. Moreover, we amortize the cost of enqueue and dequeue by inserting groups of tasks at once when possible, nominally eight in our implementation.

Local task queues are used as a local buffer for task descriptors. The local task queue is implemented as a NULL-terminated linked list. Access to the linked list is synchronized using a ticket lock [49]. A ticket lock is used to reduce contention on the L2 bus for situations where the local task queue is empty and many cores

are waiting on another core within the cluster to enqueue more tasks from the global task queue. Insertions into the local task queue, which occur when a core attempts a dequeue operation and finds the local task queue head pointer to be NULL, are synchronized by a spin lock that ensures only one core is attempting to fill the local task queue at any time; multiple global-to-local enqueuers complicate the design of barriers and could lead to increased load imbalance and contention at the L3.

The implementation of RTM relies heavily on global operations and atomic primitives provided by the architecture. Global operations are used to access global barrier state and poll head and tail pointers for the global task queue. Local atomic operations synchronize the insertion and removal of tasks at the local task queues. While global and atomic operations are seldom used directly by application code, high performance global operations are imperative for achieving low-latency enqueue, dequeue, and barrier operations. As such, the latency of global operations has a strong influence on the overall scalability of the design. As the amount of work per interval remains constant and the number of cores scales up, there is less work per core. Therefore, a higher fraction of execution is spent in the runtime executing barriers and performing task queue management relative to task execution.

Dequeue operations are split into a fast path and a slow path. For the common case where there are tasks in the local task queue and no barrier is pending, we use load-link and store-conditional operations to obtain the local task queue lock and unlink a task descriptor from the local task queue. We find this operation, including transferring task descriptor contents into registers, can be accomplished in fewer than 50 cycles, but may require many more on average if there is a high degree of contention.

If the local task queue is empty, the core that finds it empty enters the slow

path and attempts to grab the per-cluster global task queue lock. The purpose of this lock is to limit the number of cores per cluster attempting to dequeue from the global task queue to one. If the lock is held, it implies another core is accessing the global task queue and will either fill in tasks later or will notify the cores that a barrier has been reached. While the lock is held, cores waiting at the local task queue must check for more tasks by checking the value of the head pointer and the cores must check if a barrier has been reached. The cores thus alternate between checking the local barrier sense and checking the local task queue for more work. Both operations require no locks and can be performed locally at the L1 cache. When either the local task queue has tasks appended to it or a barrier is reached, the cluster-level coherence implementation invalidates the local copy of those values.

The performance of enqueue operations is critical to the scalability of RTM. If the fan-out for tasks is not fast enough, cores will starve waiting for task descriptors. If enqueue operations cannot happen fast enough, dequeued tasks will finish before new tasks are available, thus leaving cores unutilized. Similarly, if the latency of dequeue operations grows with the number of cores, scalability will be limited. We address the problem of fast enqueue by performing enqueue in parallel when possible. We can achieve this due to the SPMD execution model that ensure all threads execute the same code given certain constraints placed on synchronization and control flow. One core from each cluster is designated the enqueing core. That core determines for which section of the tasks enqueued by the user it should generate task descriptors and locally begins building up groups of task descriptors. To address the problem of fast initial dequeue, the enqueing core inserts tasks locally until a predefined number of task descriptors exists at the local task queue before it begins enqueing tasks at the global task queue. The design achieves very low overhead for enqueue operations across all our workloads.

A particularly difficult aspect of the RTM implementation is the barrier operation. The general structure of the barrier is as a reduction operation where cores enter into a cluster-level barrier and then one core from the cluster enters a tile-level barrier. One core from each tile-level barrier then participates in a global barrier. When the global barrier is reached, all cores are notified. We make use of the broadcast update operation supported by Rigel to efficiently perform the wake up. The broadcast operation on Rigel is a multicast message initiated by one core and sent from the L3 to each of the tiles where the message is replicated at each level of the interconnect down to the cores. If such a message did not exist, cores would be required to poll using global memory operations which would result in a high degree of contention at the L3. The problem would be worse when most tasks are polling and few tasks remain thus exacerbating the effect of load imbalance.

The challenge in implementing RTM barriers correctly stems from RTM supporting arbitrary enqueue between two barriers. Some BSP-like models force all enqueue operations to occur before any tasks begin; no tasks may enqueue other tasks. Fork-join models that allow for task enqueues, or forks, to occur at arbitrary points in the program have a parent-child relationship not present in RTM that allows for synchronization at post-dominators, i.e., points where all children join with their parents recursively. The implementation difficulty that this design aspect creates is that barriers must be two-phased. When there is no work pending locally, a task will check at the next level for more work. If no work is found globally, the core assumes a barrier is reached and waits. Only when all other cores have entered the barrier can the barrier proceed, as with a conventional barrier. However, there is the possibility that one of the cores not in the barrier could insert more work. Adding more work while other cores are waiting at the barrier requires that cores already in the barrier have the ability to back

out and begin dequeuing work again. The two-phase barrier is implemented by having each core waiting on the barrier check the status of the task queue and the number of other cores in the barrier, and only allowing the barrier to proceed once all cores are in the barrier and no new work is in the task queue.

3.3 Feasibility of Physical Design

Initial area and power analysis [42] shows that the design is achievable in 320 mm^2 in 45 nm technology, while consuming approximately 100 Watts. For comparison, a contemporary Intel four-core processor such as the i7-960 is 263 mm^2 with a power envelope of 130 Watts on 45 nm technology. As such, the design represents an achievable target for next generation accelerators and hybrid CPU-accelerator systems. We believe that the reasonable area and power budget for the design makes it amenable to being integrated with a CMP consisting of ILP-centric cores either by reducing the number of Rigel cores or by using future process technologies that will increase the available transistor budget.

3.4 Discussion and Summary

In summary, the Rigel design contains eight tiles, each tile contains 16 clusters, and each cluster consists of eight cores and a shared cluster cache totaling 1024 cores. The baseline architecture supports a task-based parallel programming model and a shared memory abstraction. Rigel supports these features without specialized hardware and without chip-wide hardware cache coherence. A novel approach taken by Rigel is the use of a software protocol for maintaining a coherent view of its cached memory system without the need for hardware cache coherence. The implications of this design choice are explored in great detail throughout the

remainder of this dissertation.

For the runtime and basic programming model for Rigel, a simple barrier-synchronized model with enqueue/dequeue semantics is sufficient for expressing the parallelism in our workloads. Both fast dequeue operations and scalable enqueue operations are important for supporting fine-grained tasks. Moreover, we find avoiding unnecessary blocking at the cluster by decoupling dequeue operations at different levels of the task queue hierarchy and overlapping enqueue and dequeue operations to be important to scalability. We find that our choice of allowing arbitrary enqueue operations between barriers leads to implementation difficulty due to the need to distinguish between the point when there is no work currently available, but other tasks are still executing, and the point when there is no work available and no new work will be created. Finally, we are able to provide the appearance of a single, coherent address space to application software without hardware coherence. However, we find it necessary to use global memory operations, which achieve coherence by not caching locally, to build our runtime.

CHAPTER 4

Workload Characterization

In this chapter, we provide analysis of parallel workloads running on the Rigel accelerator. We describe each of our benchmarks in detail. We characterize the runtime overheads, task granularities, and data sharing patterns of the workloads. We evaluate the cost and efficacy of coherence management in the software-managed coherence case and for hardware coherence. The patterns discussed in this chapter are used as the motivation for the architecture and coherence protocols discussed in the remainder of this work.

4.1 Workload Description

The applications that we evaluate are optimized kernels extracted from scientific and visual computing applications. The SW_{cc} variants have explicit invalidate and writeback instructions. The instructions exist in the code at both task boundaries and at barriers. The benchmarks are written using RTM, the task-based, barrier-synchronized work queue model presented in Chapter 3. We now summarize each benchmark and provide qualitative details regarding their implementation and parallel structure.

The kernels used in this work are extracted from workloads in the visual computing domain. Visual computing encompasses visualization, simulation, high-performance computing for scientific and engineering modeling, and interactive graphical computing environments. Many of these workloads are highly parallel

and scalable [12, 15], and similar workloads are cited as economically relevant for multicore processor manufacturers [81]. The benchmarks used in this work are listed below.

1. Conjugate Gradient Linear Solver (`cg`): We use a variant of the sparse linear systems solver described in [82]. We use data sets from the Harwell-Boeing sparse matrix collection. The algorithm consists of five phases that repeat until convergence criteria are met. The five phases, in program order, are a sparse matrix-vector multiply (SMVM), a reduction, a dot product of two vectors, a second reduction, and a second dot product. In our implementation, the three compute phases are parallelized and the two reductions use a hierarchically structured reduction tuned to the Rigel architecture. The key features of this benchmark are irregular task lengths, sensitivity to reduction costs, and an even mix of read-only and write-to-read-many data.

2. Dense Matrix Multiply (`dmm`): We perform a blocked single-precision dense matrix multiply on a pair of 1024×1024 matrices. Our implementation performs the multiplication in stages with barriers separating each stage. The intent is to reduce the divergence between threads to limit the working set to the size of the L3 cache. Note that a good deal of inter-thread skew is possible leading to some cache thrashing, but our technique avoids the more pathological cases of parts of the matrix being in the on-die working set simultaneously. The data structure used is an array of pointers to arrays which allows us to stagger the start of rows in memory, thus limiting conflict misses due to limited associativity at the L2. The key features of this benchmark are regular task lengths, a high ratio of read-to-write data, highly structured computation, and a relationship between working set size at multiple levels of the computation. Furthermore, reuse and sharing across

threads must be leveraged if high efficiency is to be obtained.

3. Fast-Fourier Transform (`fft`): We use a variant of the FFT proposed by Cooley and Tukey [83]. The FFT performed is a 2D FFT which consists of two phases of 1D FFTs that are separated by a transposition. The FFT phases are data-parallel and have a high floating-point operation density. The transpose stresses the interconnect due to all-to-all communication. The key features of this benchmark are high memory bandwidth and a phase that places a large strain on the interconnect.

4. Collision Detection (`gjk`): We use the Gilbert, Johnson, and Keerthi (GJK) minimum distance algorithm [84]. The algorithm computes the minimum distance between sets of convex polytopes. GJK is used in collision detection algorithms found in modeling and gaming applications. The key features of this benchmark are small and irregular task sizes.

5. 2D Stencil (`heat`): The benchmark code is derived from the heat benchmark in the Cilk benchmark suite [80]. The computation performs a stencil computation to determine the heat flow in a 2D space. The key features of this algorithm are a large degree of read-write sharing across time steps and low arithmetic intensity and little reuse for large datasets. The benchmark also makes use of double buffering for input/output datasets. One characteristic of this workload is its sensitivity to replacement policy. The high degree of read sharing at the L2 coupled with the large volume of data accessed makes proper replacement critical for avoiding unnecessary capacity misses at the L2.

6. K-means Clustering (`kmeans`): We implement a variant of the K-means clustering algorithm [85]. The algorithm takes a set of N points in a D-

dimensional space and determines K D-dimensional points that minimize the sum of the average error between the N points and K-means. The key features of this benchmark are the local and global histogramming operations.

7. Marching Cubes (`march`): We implement a variant of the marching cubes algorithm for constructing three-dimensional surfaces [86] from a 3D scalar field. This benchmark is dominated by integer operations, so the FLOPS counts presented are expected to be low.

8. Medical Image Reconstruction (`mri`): The `mri` kernel is derived from from the work of Stone et al. [87]. The key features of the benchmark are its high FLOP density and use of transcendental functions inside the inner loop. The transcendental functions can be sped up using approximate methods or specialized hardware if available.

9. Edge Detection (`sobel`): The kernel performs Sobel edge detection on a 2D image. The computation comprises a set of convolution operations. The key features of the benchmark are its potential for false sharing due to shared edge regions and sensitivity to data layout. There is also a high degree of read sharing among tasks working on a section of the image. The dimension and size of the tasks has an impact on the intra-task read sharing, while co-location of tasks that access adjacent regions of the image increases inter-task read sharing.

10. 3D stencil (`stencil`): We perform a 7-point stencil over a 3D grid of data points. The key features of this algorithm are its high memory bandwidth requirements. This benchmark is not written using RTM and instead does a static assignment of tasks to cores.

Table 4.1: Characterization of the ten workloads used in this dissertation run using a 1024-core variant of the Rigel architecture with software-managed coherence.

Benchmark	Task Length (Instructions) Min.	Max.	Mean	IPC	GFLOPS % of Peak	Cache Hit Rate (%) L1D	L1I	L2D	L3	Memory BW % of Peak
cg	189	18418	1630	0.741	2.42	0.924	0.994	0.928	0.887	8.04
dmm	25298	28682	26852	0.930	25.47	0.947	0.999	0.762	0.945	4.47
fft	2076	1944107	105669	0.397	8.61	0.864	0.999	0.788	0.419	76.34
gjk	757	84300	16027	0.694	6.30	0.882	0.997	0.948	0.865	7.11
heat	21938	71542	41207	0.695	8.88	0.903	0.998	0.846	0.506	70.29
kmeans	27721	29310	27940	0.842	16.60	0.975	0.999	0.963	0.826	1.12
march	59555	510229	141802	0.741	1.98	0.907	0.999	0.812	0.757	58.73
mri	259938	270671	265326	0.679	18.42	0.788	0.999	0.028	0.990	0.31
sobel	44045	53572	44448	0.945	22.76	0.966	0.999	0.906	0.112	12.80
stencil	–	–	–	0.810	9.51	0.844	0.999	0.792	0.572	77.01

Table 4.2: Characterization run in Table 4.1 with an on-die full-directory hardware coherence implementation.

Benchmark	Task Length (Instructions) Min.	Max.	Mean	IPC	GFLOPS % of Peak	Cache Hit Rate (%) L1D	L1I	L2D	L3	Memory BW % of Peak
cg	151	14264	1673	0.712	2.32	0.917	0.995	0.909	0.906	7.25
dmm	24491	27836	26107	0.937	27.21	0.936	0.999	0.809	0.955	4.93
fft	3431	1990425	106573	0.408	8.45	0.852	0.999	0.826	0.545	78.49
gjk	673	82917	15922	0.700	6.12	0.882	0.998	0.954	0.860	4.90
heat	19409	77783	35238	0.716	9.54	0.895	0.998	0.875	0.501	60.24
kmeans	27697	29957	27915	0.819	19.00	0.973	0.999	0.969	0.750	1.16
march	55638	502935	137052	0.760	2.04	0.895	0.999	0.829	0.783	57.72
mri	232292	248663	238419	0.583	20.44	0.762	0.999	0.560	0.979	0.37
sobel	43720	45597	43961	0.936	22.91	0.963	0.999	0.905	0.136	7.43
stencil	–	–	–	0.812	9.90	0.833	0.999	0.820	0.448	80.30

Table 4.1 provides an overview of the statistics collected on 1024-core runs of a baseline Rigel implementation. Table 4.2 shows the same statistics, but with an optimistic hardware coherence implementation that uses a full directory on-die. A full on-die directory is unrealistic due to area and power constraints, but the data demonstrates the impact of hardware coherence on performance even under highly optimistic conditions. The dataset is not a complete characterization of the workloads nor the architecture. It is meant to demonstrate some of the first-order performance issues for the design such as memory bandwidth and achieved performance. Also note that we tune the task sizes to provide the best absolute performance, which may lead to higher than expected task distribution overhead and poorer cache performance compared to configurations optimized for those metrics. More details regarding our methodology are presented in Chapter 9.

4.2 Bulk-Synchronous Patterns in Accelerator Workloads

Accelerators place different constraints on caches and coherence management relative to contemporary general-purpose CMPs. An opportunity exists, chiefly driven by characteristics of accelerator workloads, to exploit these differing constraints. However, optimizations that exploit characteristics of current workloads may come into conflict with the desire to support the variety present in emerging future workloads. To enlarge the space of applications accelerators target, they must not only support the data-parallel execution model prevalent today, but irregular task-parallel computation not well-suited to contemporary accelerators such as single-instruction multiple-data (SIMD) GPUs. Support for flexible and evolving task management models is not easily implemented with area-efficient hardware mechanisms. Therefore, we are motivated to investigate software mechanisms when possible, and general hardware mechanisms as required, for supporting a memory model tuned for accelerators.

4.2.1 Parallelism Structure

We observe that the programming styles adopted by developers for accelerator applications share a common structure, similar to bulk synchronous processing [60]. These large-scale parallel applications are composed of collections of concurrently executing tasks comprising mostly data-parallel units of work. Tasks execute between two barrier operations. The period between two barriers we define as an *interval*. The tasks exchange little or no data within an interval. At a barrier, modified shared data is made globally visible, and the next phase of computation begins.

Updates by a task during an interval can only be assumed by the programmer to be visible *after* the current interval has ended. Moreover, the programmer

cannot assume updates are invisible during the interval in which the update occurs. That is, updates are not buffered nor deferred and may take effect any time between the update and the end of the interval. Sharing modified data within an interval requires explicit programmer annotation.

We observe that popular programming models used in developing large-scale data-parallel applications do not depend on the hardware support provided by conventional systems, i.e., hardware coherence, for arbitrary sharing. In a barrier-synchronized, mostly data-parallel, task-based shared-memory programming model, coherence management is required to enable sharing; however, the *mechanisms* found in conventional CMP architectures to support arbitrary sharing through cache coherence are of marginal utility. As such, mechanisms for enabling some data, such as work queues or data structures, to be shared are required but need not cover all of memory at all times. Furthermore, the common structure present in these parallel applications is rooted in the programmer's attempt to create scalable code in a manner that is conceptually simple; thus there is minimal sharing.

4.2.2 Sharing Patterns

We provide analysis of a set of parallel visual computing workloads from VISBench [12] and from the Rigel kernel benchmark suite. VISBench consists of a set of full applications that we run on the x86 platform. Analysis of these workloads shows similar data sharing and synchronization patterns across workloads and between the two different platforms we target. Specifically, we investigate the sharing patterns of our workloads *across synchronization boundaries*. Figure 4.1 shows the naming scheme of these memory operations pictorially. Figure 4.2 gives the number of unique memory references that are shared across intervals, marked as input and output, and within an interval, marked as conflict, for VISBench

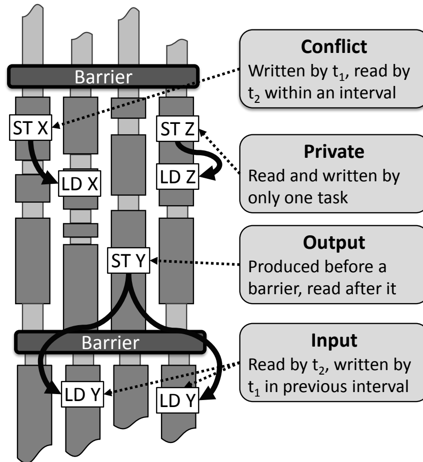


Figure 4.1: Categorization of memory access types in Rigel and VISBench benchmarks. Letters X, Y, and Z represent distinct addresses, and t_1 and t_2 represent distinct tasks.

applications, and Figure 4.3 gives the same for the Rigel benchmark suite. Note that the analysis results for MRI versions differ due to the larger degree of register spilling on x86, resulting in more private reads on x86 compared to the Rigel variant. We exclude work distribution-related sharing from results to highlight application-level characteristics.

Figures 4.2 and 4.3 give the frequency of non-private loads and stores, which are data produced by one task and consumed by one or more other tasks. Non-private accesses are further broken down into whether the values are shared between tasks within an interval, which we call *conflict* reads and writes, or across intervals, which we call *input reads* and *output writes*. The figures show that the majority of non-private loads are reads to data produced before the current interval began, i.e., input reads. At the same time, both conflict reads and writes to data shared within an interval are rare. Output writes, which are writes from one task in the current interval consumed by another task in the next interval, are more common in real applications than true shared writes which require intra-interval synchronization; moreover, true shared writes constitute a small fraction

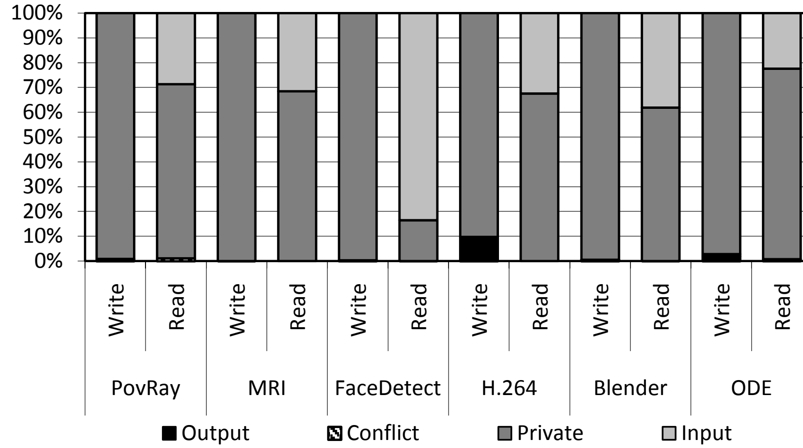


Figure 4.2: Relative fraction of memory misses for each access type for VISBench.

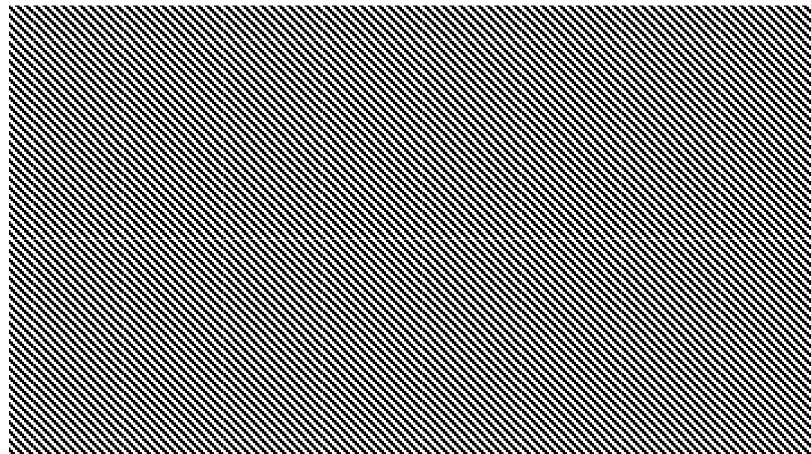


Figure 4.3: Relative fraction of memory misses for each access type for the Rigel benchmarks.

of overall execution. Also note that the number of unique output writes is much smaller than the number of input reads in the figure due to one-to-many sharing across intervals.

4.2.3 Accelerator Workload Characteristics

To summarize, we list five common characteristics in accelerator workloads:

1. Large amounts of immutable, read-shared data are present within an interval. Examples of read-shared data from our workloads include scene and

model descriptions or blocks of streaming media data.

2. Synchronization is coarse grained. Coarse-grained synchronization motivates our investigation of bulk coherence management at task boundaries. Indicative of this pattern are output writes and corresponding input reads in Figure 4.2 and Figure 4.3, which demonstrate that modified data is often read by a task *after* the interval in which the data was written ends.

3. There exist only small amounts of write-shared data within an interval. The small amount of write-shared data indicates that tasks are highly data-parallel with few data dependences between tasks within an interval. The effect is indicated in Figure 4.2 and Figure 4.3 as a lack of conflict reads and writes. Furthermore, the conflicts that do exist are structured, such as the histogramming operation on `kmeans` and reduction operations in `cg`. The structured conflicts are supported by collective operations such as atomic accumulate instructions in our workloads.

4. Fine-grained synchronization is present but rare. An example of such synchronization is atomic updates to shared data structures. Although not shown in the figures directly, we observe that much of the fine-grained synchronization that we do find is used for task management and not for application code.

5. When write sharing within an interval does exist, it is usually between few sharers.

Collectively, these characteristics demonstrate that little coherence management is required within an interval, indicating the potential for pushing coherence management into software to be performed logically at the end of an interval. At the same time, mechanisms must be present to allow small amounts of fine-grained

synchronization and data sharing within an interval for supporting task management efficiently. Our findings further motivate the use of shared caches that can amortize the costs associated with data access to read-shared data, a prevalent access pattern in our target workloads.

4.2.4 Cache Coherence Management

A mechanism for maintaining coherence, whether fully supported in hardware or a software protocol partially supported by hardware, cannot simply be omitted from the design of future accelerators, but the constraints placed on accelerators with respect to coherence differ from those of general-purpose CMPs. CMPs rely on cache coherence and global synchronization mechanisms to provide shared resource management. Alternatively, an accelerator architecture can employ relaxed memory models, explicit local and global memory operations, and a task-based programming model to execute the coherence actions needed to enforce the memory model at barriers, thus providing structure without sacrificing performance. As a substitute for hardware cache coherence, we investigate the use of software enforcement of our Task-Centric Memory Model, as described in Chapter 5. In the next section, we provide a more detailed overview of coherence management costs for accelerators.

4.3 Coherence Overhead for Accelerator Workloads

In this section we discuss coherence overheads for accelerator workloads when using hardware coherence and software-managed coherence. We evaluate the additional network traffic and the efficiency of cache management instructions executed by software-managed coherence.

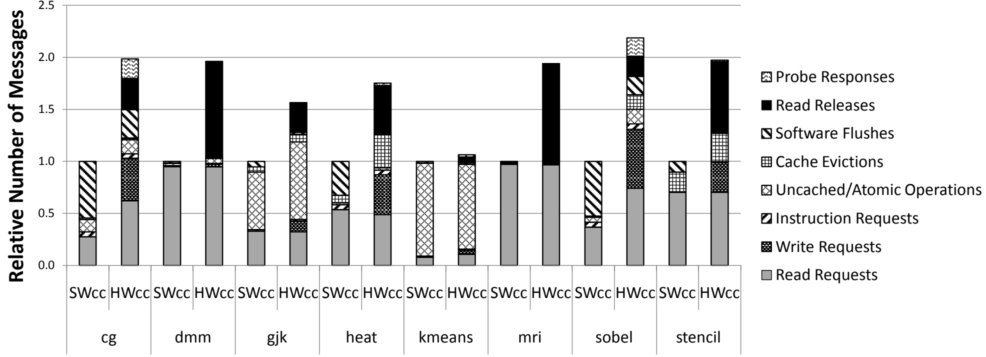


Figure 4.4: Message count from the L2 cache to the shared L3 normalized to software coherence (SW_{cc}) results. Note that HW_{cc} contains some software flushes as an optimization for data known to be no longer needed in the L2.

4.3.1 Network Traffic

We simulate the Rigel benchmark suite running on a 1024-core Rigel system. The details of the simulation methodology are further described in Chapter 9. Figure 4.4 gives the messaging overhead for an optimistic implementation of hardware coherence (HW_{cc}). For this experiment, we assume a complete full-map on-die directory with infinite capacity and full associativity. We choose this aggressive and potentially unrealizable design point since it eliminates broadcasts and capacity or conflict evictions at the directory. The design point thus represents an optimistic picture of the overhead of directory-based hardware cache coherence.

The software-managed protocol (SW_{cc}) uses explicit flush and invalidation instructions inserted into the software to manage coherence. Moreover, SW_{cc} reduces instruction stream efficiency since it must issue explicit cache flush instructions, which show up as additional software flush messages in Figure 4.4.

Figure 4.4 indicates markedly increased message traffic across all benchmarks for HW_{cc} except for `kmeans`, which is dominated by atomic read-modify-write histogramming operations that are simulated similarly for both SW_{cc} and HW_{cc}. The additional messages come primarily from two sources: write misses and read release/invalidation operations. For SW_{cc}, software is responsible for tracking own-

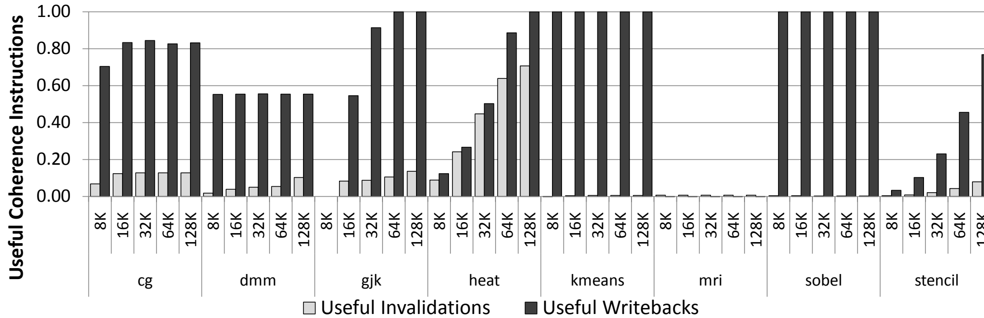


Figure 4.5: Writeback and invalidate efficiency for different L2 cache sizes. An inefficient access is one that is performed by software to a line no longer present in the cache. For all of our other experiments we use a 64 kB cache.

ership. For a non-inclusive cache hierarchy [1,64] per-word dirty and valid bits are maintained in cache. Writes can be issued as write-allocates under SW_{cc} without waiting on a directory response. Under HW_{cc} we do not support silent evictions, and thus read releases are issued to the directory when a clean line is evicted from a local cache (L2). For SW_{cc}, there is no need to send any message, and the invalidation occurs locally. Even if a protocol without read releases were used, HW_{cc} would still show significant message overhead for invalidation cache probes; the implications of a protocol without read releases is discussed in a later section.

4.3.2 Software-Managed Coherence Efficiency

Figure 4.5 evaluates the efficiency of issued flush and invalidate instructions under SW_{cc} by counting the number of such instructions that operate on valid lines in the cache. Under a software protocol with deferred coherence actions, the state of lines must be tracked in memory, which can be less efficient than maintaining a small number of bits with each cache tag, which is the conventional method for tracking coherence state. Furthermore, flush instructions can be wasteful, as the lines they target may have already been evicted from the cache by the time the SW_{cc} actions occur. We show this effect quantitatively in Figure 4.5 where we

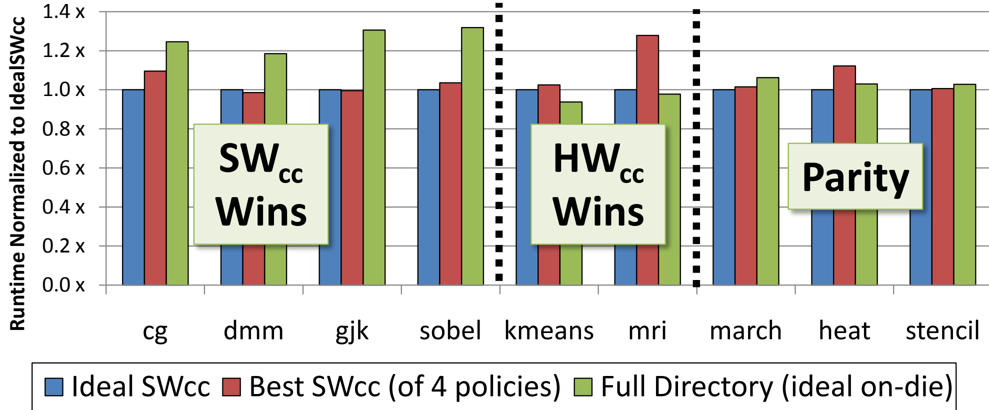


Figure 4.6: Runtime of idealized hardware and software-managed coherence compared to the best case implementation using the Task-Centric memory model.

measure the number of SW_{cc} actions that are performed to lines valid in the L2. Indeed, many of the coherence instructions issued dynamically are superfluous, operating on lines not present in the cache.

4.3.3 Performance Trade-Offs in Coherence Management

Figure 4.6 compares the runtime of our workloads under hardware-managed and software-managed coherence. The hardware-managed coherence implementation is a full on-die directory which allows us to evaluate best-case directory performance. We also include an ideal software-managed coherence implementation that removes all unnecessary writeback operations to evaluate optimistic performance for a software-managed scheme. Note that it is possible for the optimistic variant to be slower than realistic SW_{cc}due to network reordering, additional contention due to self-synchronization of tasks, and variation in task scheduling. The figure shows that no one policy provides a clear performance advantage, suggesting that a mix of policies is necessary to achieve optimal performance. Moreover, the programmability implications of moving from one method to another might make one model more advantageous than another.

4.4 Discussion and Summary

In this chapter we have described the applications we evaluate in this dissertation. We discussed the bulk-synchronous processing pattern common to accelerator workloads. Using this structure, we described the prevalent data sharing patterns in accelerator workloads and how they can be exploited by a memory model tuned to accelerators. We evaluated the overhead of coherence management for accelerators and found that from a performance perspective, neither software-manged coherence nor hardware-managed coherence is always the best choice. In the case of software-managed coherence, conservative actions necessary to maintain correctness when hardware caches are in use can lead to high instruction stream overhead due to unnecessary cache management operations. Hardware-managed coherence suffers from an unnecessary increase in network messages and on-die coherence state storage. The unnecessary increases have two causes. One is that the data is unshared, thus obviating the need for coherence. The other is that the workload fits the BSP pattern well and has a predictable sharing pattern where software management overhead would be minimal.

CHAPTER 5

Task-Centric Memory Model

In this chapter, we present the Task-Centric Memory Model (TCMM) for 1000-core compute accelerators. Visual computing applications, examples of which are discussed in Chapter 4, are emerging as an important class of workloads that can exploit 1000-core processors. In these workloads, we observe data sharing and communication patterns that can be leveraged in the design of memory systems for future 1000-core processors. Based on these insights, we propose a memory model that uses a software protocol, working in collaboration with hardware caches, to maintain a coherent, single-address space view of memory without the need for hardware coherence support. An abbreviated form of this work was previously published in [77] and [88].

5.1 Design

Our baseline system is a multicore processor with a single address space and hardware caches, but without hardware cache coherence. In the absence of hardware support for coherence in a single address space multicore processor, a software-defined memory model is necessary to achieve consistent behavior for cases where data may need to be shared between threads executing on different cores. A share-nothing approach and explicitly managed address spaces are ways to avoid the coherence problem inherent to CMPs with a single address space; similar approaches are taken on the Cell [57] and GPUs [4]. Our goal, however, is to address

the coherence problem using scalable hardware while leveraging the characteristics of scalable applications.

The distinction between the multiple address space paradigm for accelerators, e.g., GPUs and Cell, and the single address space paradigm for accelerators, e.g., Rigel, is pertinent to this dissertation. While the former lends well to SIMD execution, scratchpad memories, and explicit software management of locality, the latter is more amenable to MIMD execution, hardware caches, and hardware management of locality. The motivation for why these features are beneficial for SIMD and MIMD, respectively, is discussed in greater detail in the motivation in Chapter 2. While SIMD accelerators with software-managed local memories are prevalent, we assert that the limiting factor for wide-scale adoption of cached MIMD accelerators is the lack of a scalable, low-complexity means to solve the coherence problem for 1000-core processors similar to Rigel. This section presents one such solution.

In this section, we discuss the design of the software-defined memory model used by the baseline Rigel processor. In our approach, we leverage the bulk-synchronous structure of many parallel applications as well as the implications for sharing patterns of this structure, as illustrated in Chapter 4, to develop the Task-Centric Memory Model. By having the programmer reason about memory blocks that are read-only, shared, or private during each interval, the memory model we define allows software to achieve the behavior of a coherent design without hardware cache coherence. The goal is to achieve the semantics of a shared single address space without the need for the complex hardware and performance overheads associated with building a cache coherent CMP. The problems related to scaling hardware cache coherence that we address include: the need for ordered and/or complex interconnects, false sharing, coherence state storage area overheads, e.g., directories or snoop filters, and the difficulty inherent to eliminat-

ing microarchitectural races from hardware coherence protocols.

The Task-Centric Memory Model defines a *coherence domain* as a logical grouping of memory blocks for which coherence guarantees are provided collectively by the memory model. The model requires that software performs the necessary actions to transition blocks between the different domains during program execution. Software in this case refers to either a developer explicitly managing sharing, a library that hides the transitions from the developers, or a tool, such as a compiler, that inserts operations to orchestrate the transitions automatically. Once a block is moved into a state other than the initial (`clean`) state during an interval, it cannot transition to another coherence domain until after a global synchronization point is reached; this constraint is relaxed later, but serves as a conservative simplifying assumption for the model.

5.1.1 Coherence Algorithm

A block of memory follows the state machine diagrammed in Figure 5.1. Tasks t_i operate on blocks using the following memory instructions: local loads (`L.LD`), local stores (`L.ST`), global loads (`G.LD`), global stores (`G.ST`), writeback operations (`WB`), and invalidate operations (`INV`). Writeback operations write a line back to the global (L3) cache if present in the cluster (L2) cache and mark the line unmodified at the cluster cache. Invalidate operations make a line invalid in the cluster cache if present. Note that in our implementation, L1 caches are transparent to software, so the coherence algorithm need not consider their state. Each set T_x represents the collection of tasks sharing a block in a particular state: `clean` (T_C), `globally coherent` (T_G), `immutable` (T_I), and `private` (T_P). The private domain is broken into two states for clarity. The two states also enable optimizations since a clean line in the `private` state can simply be dropped when an invalidation is issued

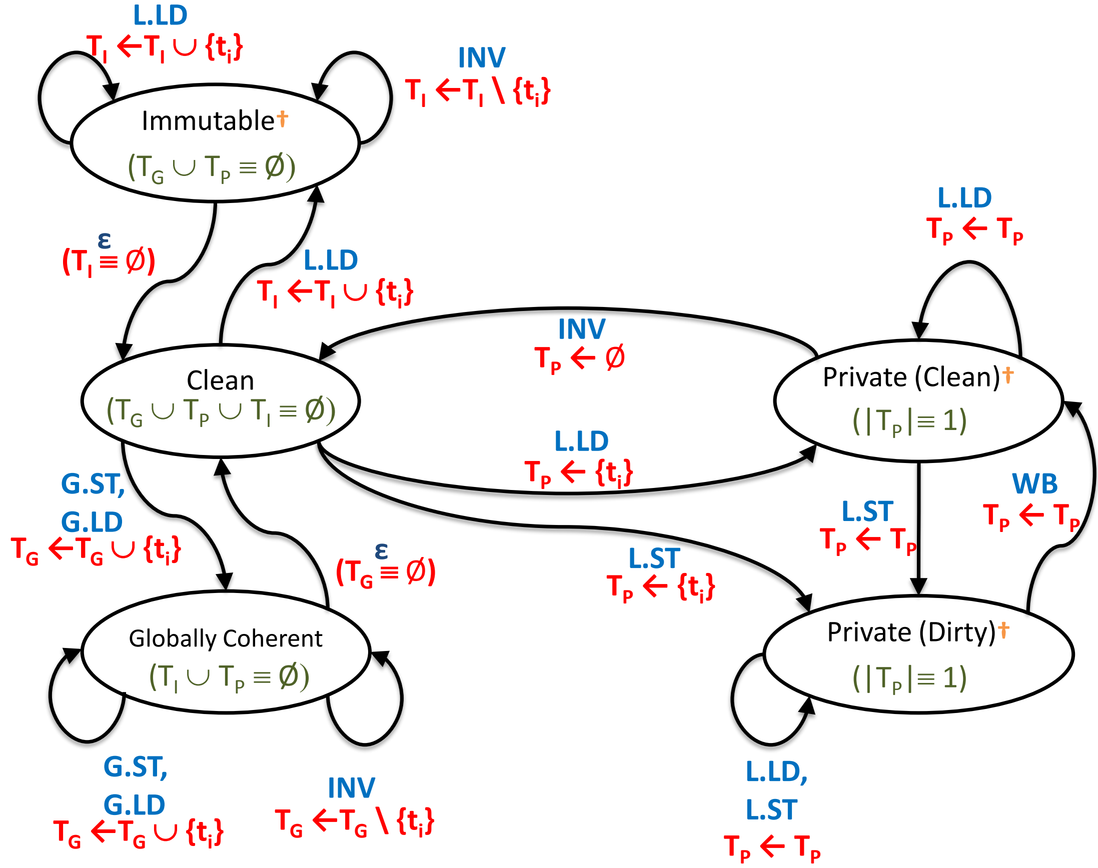


Figure 5.1: State transitions for memory blocks in the Task-Centric Memory Model. Actions include: Local loads (L.LD), local stores (L.ST), global loads (G.LD), global stores (G.ST), write backs to the global cache (WB), and cluster cache invalidates (INV). The † notes states that may cache a block at the cluster cache. The set of tasks sharing a block in state X is denoted by T_X. Any transition absent from the diagram is disallowed by the model.

while a writeback in the `private` state requires communication with the L3.

There are six properties defined for the memory model:

1. All blocks start in the `clean` state. $(\emptyset \equiv T_I \cup T_G \cup T_P | time = 0)$

2. A barrier is a global point of synchronization. All memory operations performed before a barrier must be complete before any processor leaves the barrier.

3. Blocks may only transition between accessible states by first passing through the `clean` state and after a barrier is reached.

4. A block may be in exactly one (non-`clean`) accessible state from the perspective of all cores in the system at any time. $((\emptyset \equiv T_I \cap T_G) \wedge (\emptyset \equiv T_I \cap T_P) \wedge (\emptyset \equiv T_G \cap T_P))$

5. A block in the `private` state must have $\|T_P\| \equiv 1$.

6. Loads (`G.LD`) that target a block in the `globally coherent` state return the last write to that location. All cores in the system see the same ordering of updates to that location, i.e., the block is kept coherent.

The software coherence protocol must interact properly with the underlying hardware to ensure correct execution. For instance, the `private (clean)` state corresponds to a data value in the cluster cache that does not have its dirty bit set. The cluster cache controller may invalidate the line on an eviction, implicitly moving the line into the `clean` state. Should the core previously holding the block in the `private` state reissue a load to that location, the cluster cache controller must fetch the value from the global cache. The value is guaranteed to return the same value as if the eviction had not occurred since, by properties 4 and 5 above, ownership of the block is held solely by the core issuing the load. Global

atomics are restricted to be only performed on `globally coherent` blocks and have the same semantics from the perspective of the memory model as global loads and stores. Having two distinct cluster caches hold the same block in the dirty state represents a race condition that is possible in hardware, but is disallowed by software that obeys the memory model defined above.

5.1.2 Memory Ordering

Ordering of memory operations is defined separately for operations performed within distinct coherence domains. Ordering must be defined when *conflicting* accesses exist. A conflict is defined as at least two cores accessing the same block with at least one access being a write. Blocks in the `clean` and `immutable` states can never have conflicting accesses by definition. Blocks in the `clean` and `immutable` states have a single value that is visible to all cores.

Property 5 of the memory model ensures that updates to `private` blocks are only ever visible to a single core and therefore by definition no conflicting accesses may occur. Loads by a core return the last store to the block performed by the core while in the `private` state or, if the block has not been written by the core since becoming `private`, the value of the block when it was in the `clean` state is returned. Blocks in the `private` state therefore need only respect dependences implied by program order. Access by more than one core to `private` blocks is disallowed by the model.

Conflicts may occur for blocks in the `globally coherent` state. Blocks in the `globally coherent` state are kept cache coherent. We define the ordering of all accesses to all blocks in the `globally coherent` state to conform to processor consistency [89]. Processor consistency states that all memory operations from any one core appear in the same order to all other cores, but different cores can see

the different interleavings of accesses from two other cores. Processor consistency is weaker than sequential consistency [90], which defines a single global ordering of memory operations. However, processor consistency allows for optimizations not possible with a stricter model [91, 92].

A stricter model for `globally coherent` data is required to enable accesses to that data to be used as synchronization primitives when necessary. When used as synchronization variables, `globally coherent` data can impose a partial order on memory operations across coherence domains. A global ordering of accesses is defined at barriers by property 2. For implementation and optimization reasons, the memory model defines ordering between dependent operations that *cross coherence domains* from a single core similarly to weakly consistent models. The memory model defines that reads to `private` blocks followed by writes to `globally coherent` blocks from a single core respect program order. Reads to `immutable` or `globally coherent` blocks followed by writes to `private` blocks from a single core respect program order. Other orderings across cores and coherence domains are undefined by the model. A memory fence operation is provided by the ISA to ensure that all memory operations, including writebacks and invalidates, issued by a core executing the fence complete before the fence retires. No new memory operation may be initiated until after the fence has retired. A global memory fence can be constructed by having all cores issuing a memory fence prior to entering a global synchronization barrier.

5.2 Optimizations

The Task-Centric Memory Model is able to provide the appearance of a coherent single address space on a chip multiprocessor without hardware cache coherence. However, strict adherence to the model unnecessarily limits optimizations.

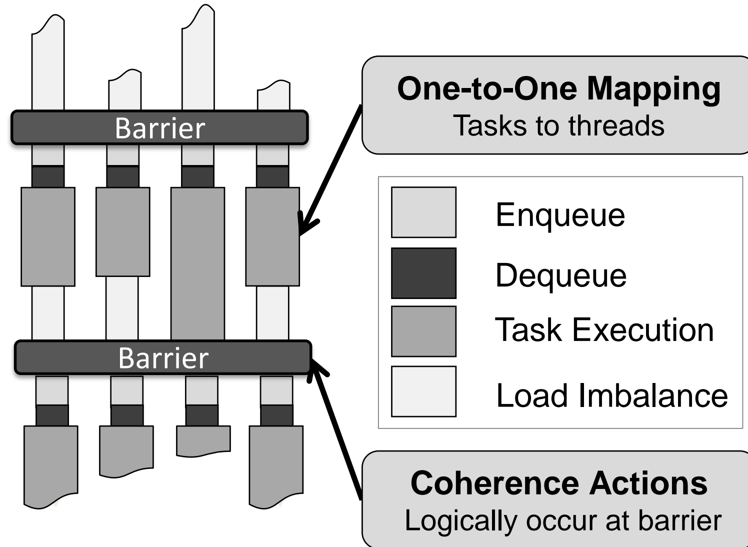


Figure 5.2: Logical flow of tasks and coherence actions in the Task-Centric Memory Model.

The baseline model limits software and hardware prefetching capabilities. The model forces inter-task shared accesses that occur within an interval to make use of the `globally coherent` state and thus to conservatively access high-latency global caches on all accesses to that data throughout the interval. The `globally coherent` state is particularly relevant to the produce-consumer sharing pattern that does not lend well to cross-barrier synchronization. Forcing all data to the `clean` state also requires aggressive invalidation and cache flushing that is unnecessary in many cases. A simple example is stack data that is allocated to each core once and ownership, and by extension its classification as `private` data, will not change across barriers. By extending the baseline Task-Centric Memory Model presented, many of these limitations can be addressed.

We evaluate different policies for deciding when to perform coherence actions *before* the end of an interval. In the baseline model, all tasks are considered independent and all explicit coherence actions are performed at interval boundaries as shown in Figure 5.2. Further optimization can be performed by taking a *thread-*

centric view of coherence management, i.e., a view that considers the sequence of all tasks run on a single core within an interval as one unit for which to schedule coherence actions instead of at the completion of tasks. As an example, we can weaken property 3 by adding: Not all blocks need to be clean at barriers, only those that undergo state transition across the barrier. While the general problem of determining what data may be made coherent lazily is difficult, there are opportunities to exploit always-private data, such as stack allocations, and programmer assertions for immutable data, such as the `const` keyword in the C programming langauge.

When available, locality can be exploited by augmenting an underlying assumption of the model that a task maps to a single core. Optimization can be performed using cluster-level sharing by extending the model to map groups of tasks to clusters instead of a single task to a core. By reconsidering the level at which work is mapped to execution resources, a first level of shared cache, such as the cluster cache on Rigel, can now be used as the point of coherence for data. Doing so allows for data that would otherwise be required to exist in the `globally coherent` state, thus suffering high latency to access the furthest level of the hierarchy, to be effectively privatized to more local caches when all tasks accessing the data can be co-located as part of a group.

The Task-Centric Memory Model supports staged porting of applications initially developed assuming full hardware cache coherence and porting efforts starting from a sequential implementation. To do so, initially the `globally coherent` state is used for all data in the application to provide the appearance that all data is kept coherent at all times. We call this the *debug model*. While a performance penalty is paid for using the `globally coherent` state for all data due to the restrictions on local caching, the assumption of coherence holds, and thus enables correctness for ported software. The performance penalty on Rigel is attributed

to the order of magnitude reduction in cache bandwidth, i.e., L2s with 256 ports versus L3s with 32, and the order of magnitude increase in latency due to L2 to L3 network traversals on every load and store. Queueing and contention in the network magnifies the costs. However, even at a 10×–100× slowdown, the debug model is still orders of magnitude faster than simulation without any discrepancies in execution between the production system and the simulator.

With a correct implementation on the new platform to serve as a baseline, software can be modified to make use of other states in the memory model to improve performance by relaxing coherence guarantees as needed. Supporting both a strict and weak coherence model in one platform allows programmers to achieve the benefit of a 1000-core processor during development while also achieving scalable results in production.

5.3 Coherence Management Placement

A naïve implementation of the memory model, which strictly adheres to task-centric actions, requires a large number of writebacks and invalidates to occur at task boundaries. The added memory traffic at the start and end of each task may lead to poor bandwidth utilization. Due to queuing delays in the network and at the memory controller, the latency for memory operations during these periods also grows precipitously. Lastly, the read sharing benefit of immutable data is decreased if shared data are aggressively invalidated from the shared cluster caches. For data that does not change state intra-interval, coherence actions are unnecessary within the interval. The combination of these effects leads us to explore alternative policies for scheduling coherence actions.

Coherence actions need not occur at task boundaries. Figure 5.3 illustrates different locations where coherence actions may be placed relative to task execu-

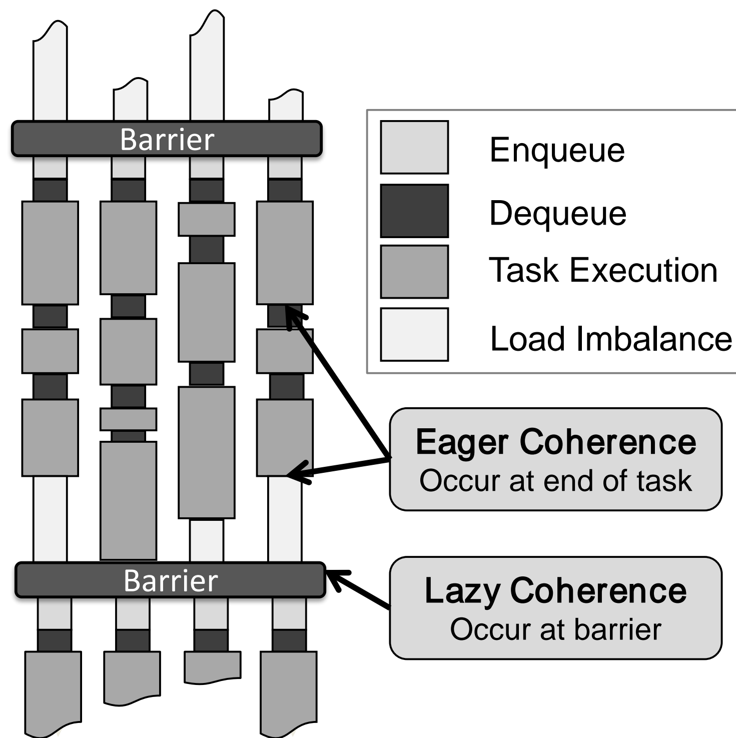


Figure 5.3: Choice of coherence action locations using the Task-Centric Memory Model.

tion. Coherence actions can be deferred by the runtime as long as state changes that occur across a barrier are completed by the end of the interval. To that end, we evaluate combinations of two policies, *lazy* and *eager*, for the writeback and invalidate components of coherence management. In our evaluation, we use an optimistic baseline that mimics the effects of write-update hardware coherence with zero-cost updates between cluster caches; no software coherence actions are taken in the baseline. Lazy actions occur en masse at barriers and eager actions occur at task boundaries. The four policies we implement are eager invalidate-eager writeback (EIEW), lazy invalidate-eager writeback (LIEW), eager invalidate-lazy writeback (EILW), and lazy invalidate-lazy writeback (LILW) relative to the optimistic baseline. We evaluate each of these policies in Chapter 9.

More generally, there is a spectrum of eagerness that we can employ in the TCMM. We list the advantages and disadvantages of each level in Table 5.1. The most coarse, or lazy, form of coherence actions is to place them after the barrier has executed but before the next interval begins. The caches are flushed when a barrier is reached. The benefit of this approach is that no state tracking must be done within the interval and the instruction overhead is minimal. However, such an approach unnecessarily removes still valid data from the caches and creates a bursty load on the network, dilating the barrier in time.

At the other extreme end of the spectrum, we could elect to implement the TCMM by performing all coherence actions immediately following any shared read or write. This represents the most eager form of coherence management placement. The benefit is that the compiler could trivially insert invalidates and writebacks without any assistance from the runtime. With more complex analysis, it might be possible to eliminate all but the last access to a shared word and remove coherence actions from all persistently private data. The disadvantage is the high coherence management overhead since the loads and stores are word granularity,

Table 5.1: Comparative advantages and disadvantages for different coherence action placements.

Placement	**Advantages**	**Disadvantages**	**Comments**
Every `ld/st`	Easily automated; Zero overhead for tracking	High network overhead; Low cache utilization for shared data	Implementable in the compiler [93]
Basic block/function	Automatable; Avoids excessive use of global operations	Limited scope may preclude optimizations	Avoids inter-procedural analysis
Task Boundary	Allows cores to overlap coherence actions with task execution; may evict useless data	Requires programming model with restrictions on data accesses to be amenable to automation	Eager approach evaluated in this work
Thread Boundary	Allows inter-task read sharing; Overlaps coherence actions when thread length is irregular	Does not overlap writebacks from one task with the next	Lazy approach evaluated in this work
In-bulk at Barrier	Trivial implementation; Low overhead for tracking	Fails to exploit inter-interval sharing; Does not overlap communication and computation	Additional tracking hardware could accelerate flushes and reduce unnecessary invalidations

but invalidations and writebacks can be performed once for an entire line.

With our choice of end-of-task and end-of-thread for instrumenting code with eager and lazy coherence actions, respectively, we believe we achieve most of the benefits of more aggressive policies without the additional complexity of compiler-driven analysis and instrumentation. We leave such investigation to future work.

5.4 Summary and Discussion

In this section we have motivated and described the Task-Centric Memory Model. The goal of TCMM is to achieve the illusion of a shared single address space for applications that use a task queue-based programming model synchronized by barriers. We evaluate the sharing patterns in a representative selection of applications developed using such a programming model. Our analysis demonstrates that a large fraction of write-to-read sharing is performed across barriers. The TCMM exploits this fact to implement coherence actions in software logically at barriers.

CHAPTER 6

Scalable Probe Filtering

In this chapter we describe a probe filtering coherence implementation and an extension to the scheme, scalable probe filtering (SPF), to increase scalability. The purpose of evaluating probe filtering is to better understand the benefits and limitations of similar schemes used in CMPs today [94, 95]. We assume an architecture with a shared last-level cache (LLC), common in many accelerators [4, 14, 57, 96] and throughput-oriented architectures [2, 97]. If no coherence information is tracked at the LLC, all coherence actions require broadcasts to determine the sharing state of the line. Fundamentally, a probe filter attempts to capture recent coherence requests at the LLC in a cache-like structure to avoid broadcasts when possible.

Duplicate tag schemes [64] and complete sparse directories eliminate all broadcasts by tracking complete sharing information at the LLC, but require high associativity and capacity. Subset sparse directories that drop entries on eviction, which are in effect *probe filters*, can eliminate a large fraction of broadcasts with smaller and less associative structures. Tracking subsets of coherence information has also been used to accelerate SMP coherence protocols by reducing LLC tag accesses [63] and reducing the number of coherence probes [62]. In our scheme, we keep a cache of sharing vectors, called the probe filter cache (PFC) to reduce the frequency of on-chip broadcasts. We show that a PFC increases scalability of broadcast schemes, but it is insufficient for scaling to a thousand cores.

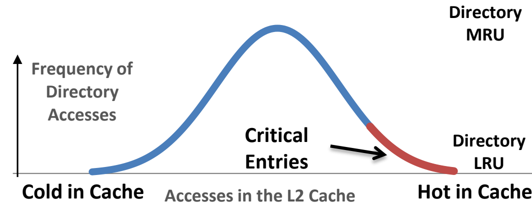


Figure 6.1: Illustration of the difficulty in selecting a directory replacement policy due to the information disconnect between L2 caches and the directory.

6.1 Baseline Probe Filter Architecture

A PFC is basically a cache of directory entries. Implementing a PFC enables accesses that hit in the PFC to be completed without broadcasting. Figure 6.1 shows the relationship between access frequency at the LLC or directory and access at the L2 cache, close to the cores. The PFC reconstructs evicted directory entries the next time the entry is allocated, rather than invalidating all copies on eviction, to avoid needless invalidations for read-shared data, as represented on the right side of Figure 6.1. The PFC broadcasts probes on each PFC miss and uses the L2 responses to reconstruct the sharing vector.

The probe filter performs well for two common patterns in our workloads. One pattern of directory access is data that adds and remove sharers frequently, due to the true sharing pattern or frequent L2 evictions and re-requests. These PFC entries, shown in the middle of Figure 6.1, will typically stay in the MRU position and will not be evicted if an LRU policy is used. Based on our analysis, we select an LRU policy for this dissertation. There is a potential for adaptive policies that address variations in workload, but an LRU policy was found to be sufficient for our purposes. Another possible mechanism would use sideband information sent from the L2 to the directory. The out-of-band information would indicate the hottest lines in the L2 which would in turn indicate to the directory which lines it should avoid evicting.

The second pattern is data that is requested once and remains in the L2s for a long period of time, due to frequent accesses by the cores. It is shown on the right of Figure 6.1. These entries will become cold in the PFC and be evicted. However, since PFC evictions do not invalidate the entries at the L2s, the cores can continue to access the lines and performance is not adversely affected.

6.2 Scalable Probe Filtering Architecture

Probe filtering's performance is limited by the broadcasts required on all PFC misses. PFC misses are common for workloads with large datasets and those with a large fraction of touch-once data. For large working sets, or datasets with imbalanced access patterns where LLC accesses are biased toward certain PFC banks, the PFC is unable to capture shared entries long enough to be effective. The naïve solution is to enlarge the PFC, but that comes at a prohibitive cost in area and power, as we will show in our evaluation. Data that is accessed only once at the PFC and considered touch-once at the LLC may be reused heavily at higher levels of the cache, i.e., the L1 and L2 caches. The application is not memory-bound as it may have good cache hit rates at the L2, but every new access at the PFC misses, causing touch-once data to continually evict potentially useful shared lines.

We investigate scalable probe filtering whereby a high-priority network for broadcast-collective operations and additional logic in the routers between the L2 and LLC are added to the PFC to accelerate broadcast probes. While the SPF implementation is tailored to our baseline architecture, it exploits the general design features of a high-bandwidth cache hierarchy and interconnect tuned for throughput-oriented workloads. In such a configuration, the goal is to maximize memory bandwidth and route data from the LLC and DRAM to the cores.

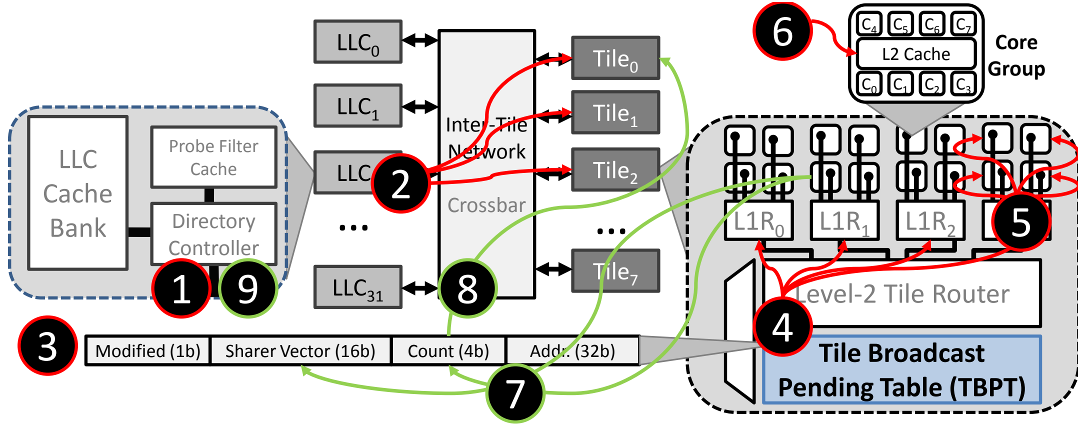


Figure 6.2: Probe filter block diagram.

The network is not optimized for arbitrary core-to-core sharing. SPF could be extended to support arbitrary topologies and system architectures, but we focus on throughput-oriented CMP topologies, similar to those found in GPU-like architectures, because throughput-oriented workloads are the focus of this work.

A broadcast probe consists of two phases and is illustrated in Figure 6.2. We describe the process in greater detail in the paragraphs that follow.

6.2.1 Broadcast

When a probe filter miss occurs and a broadcast is required (1), the directory controller injects one message into the crossbar for each of the tiles (2). The message is a BCastInvReq if the request missing in the PFC is a write request, and a BCastShareReq if it is a read request. Each PFC supports a finite number of concurrent broadcasts, but PFC hits can be serviced under misses. For this dissertation, we assume a single outstanding broadcast per PFC bank, or 32 total outstanding broadcasts. We evaluated allowing more outstanding broadcasts, but found the marginal benefit to be negligible.

When a broadcast arrives at the root of the tile interconnect, an entry is inserted into the *tile broadcast pending table* (TBPT) (3). The TBPT is a small

set-associative structure with a set for each PFC bank and a path for each outstanding request allowed per PFC bank. The set is indexed based on a subset of the bits in the address. Each entry consists of a bit vector to record the ACK or NAK L2 responses, a bit to denote whether or not the line was detected as modified at one of the L2s, and a count of received responses. In our design, a 32-entry direct-mapped TBPT is used, totaling 212 bytes per tile or 1696 bytes for the entire chip. When an entry is inserted, the bit vector is cleared and the counter is reset. The incoming message is then replicated on each outgoing port of each router ④ until each L2 receives a broadcast probe ⑤.

6.2.2 Collection

Each L2 responds to a BCastInvReq by invalidating the line, if present, and sending a BCastInvRep ⑥. A BCastShareReq results in either a BCastShareAck if the L2 shares the line, or a BCastShareNak otherwise. If the line is dirty in the cache, the L2 is the sole owner and must source the data. In this case, the L2 then issues two response messages: a writeback containing the data and a BCastInvWB to notify the PFC to wait for the data.

When a broadcast reply arrives at the root of the tile interconnect, the TBPT entry is updated by incrementing its counter and setting the L2's corresponding bit if the reply is an ACK ⑦. In response to receiving a BCastInvWB, the TBPT will also set the modified bit in the TBPT entry for the line. When all responses are collected, indicated by the counter saturating, the TBPT generates a response message for the tile that includes the sharing vector and the modified bit. The TBPT sends the message to the PFC bank that initiated the broadcast ⑧. The PFC collects the TBPT responses and merges them into a single reconstructed directory entry. When all broadcasts have been collected, the pending request

is serviced and the reconstructed entry is inserted into the PFC (9). If the line corresponding to the PFC miss is modified in one of the L2s, the PFC will not service the missing request until all broadcast replies and the writeback have arrived. Note that no ordering is required between the broadcast replies and the writeback since the directory must wait for both before proceeding.

As an optimization, the response could be sent as soon as the writeback arrived at the PFC. However, that would allow transient hardware state to cross a protocol state change, potentially greatly complicating the protocol implementation.

6.3 Summary and Discussion

The probe filter cache is an approach to scalable coherence that is similar to existing CMP implementations. We add greater scalability to a baseline PFC by using what we call the scalable probe filter. The SPF adds a small amount of support to the network for broadcast and collective operations. The benefits of our additions include increased broadcast throughput when invalidations are needed and accelerated collection of responses. Without an SPF, these operations can take time linear in the number of nodes. With SPF, the time to complete broadcasts and collections becomes the logarithm of the number of nodes. The linear-log trade-off is also true for the aggregate number of invalidation messages sent, which would otherwise compete with other network transactions.

The SPF leverages the hierarchical nature of our baseline CMP design. While we believe that a hierarchy in the network will be a common design pattern in future CMPs, the SPF could be adapted to other topologies by placing more complexity in the network and distributing the PFC banks in the case of a distributed NUCA cache. However, the SPF may be a suboptimal design choice for these systems if the load on the network cannot be easily reduced and the

complexity in the routers grows too high. Regardless, we find that the SPF is a reasonable approach to large CMPs, and we demonstrate its ability to scale using an architecture similar to those of CMP and accelerators available today.

As our results will demonstrate, the SPF is more scalable than a PFC alone. However, the SPF approach has pathologies that hinder performance for common data access patterns found in accelerator workloads. Large amounts of touch-once data and large working sets are not captured by a reasonably sized PFC. While the SPF improves performance under these conditions, it still fails to achieve scalability to 1024 cores across all workloads. While a CMP equipped with SPF could scale to perhaps 100 cores, we find it inadequate for achieving our goal of 1024-core scalability. As such, in the next chapter we present another hardware coherence mechanism that addresses the PFC sizing limitations of SPF.

CHAPTER 7

Waypoint

In this section we present WayPoint, a lightweight directory coherence architecture for chip multiprocessors. Our evaluation of a 1024-core CMP demonstrates that banked, set-associative on-die directory caches provide a viable alternative to full directories. Moreover, on-die full directories would require high associativity and capacity to be effective, making WayPoint a viable option for hardware coherence at 1024 cores. Analysis of parallel workloads shows that directory cache associativity and capacity demands vary in time, across sets, and between benchmarks, requiring over-provisioning to avoid conflicts at the directory.

To address these issues, we present WayPoint, a coherence architecture for handling associativity overflow conditions by evicting conflicting directory cache entries into a next-level cache stored in cached system memory. WayPoint enables the implementation of a scalable coherence protocol with low hardware complexity and relaxed network ordering requirements. Furthermore, WayPoint enables a non-inclusive, non-exclusive cache hierarchy that decouples the associativity and sizing requirements at different levels of the cache—a key problem for CMPs with many independent first-level caches.

7.1 Motivation

We motivate the design of WayPoint by examining data sharing behavior of data-parallel kernels. We investigate the varying requirements placed on the as-

sociativity and capacity of directory caches over time and give insight into issues related to directory replacement policy.

7.1.1 Sharer Tracking

Two prevalent mechanisms for tracking sharer state are (1) pointer-based limited directories, which generally require $s \log_2 p$ bits per entry for tracking s sharers in a system with p potential sharers, and (2) full-map directories, which require p bits. A limited directory can be implemented with less storage for up to $\frac{p}{\log_2 p}$ tracked sharers. If the directory is sparse, tag overhead must also be considered.

For highly scaled CMPs with hundreds of cores or more, workloads will be dominated by data-level parallelism. Data-parallel applications tend to exhibit large amounts of read-shared data. Writes to this data do occur, but they are infrequent. Furthermore, fine-grained synchronization and data sharing are uncommon in these applications [12, 98]. We have evaluated both limited and full-map directories. Our results show that limited directories impact performance minimally for most workloads when compared to full directories. For consistency and to provide a more scalable implementation, we evaluate a limited directory with $s = 4$ in Chapter 9.

7.1.2 Directory Size

A system with p sharers, where each sharer has a w-way, l-set cache, may cache up to $p \times w \times l$ distinct cache lines at a time. A simple directory scheme that tracks all cached lines would therefore at a minimum require a sharer vector or pointer list, t tag bits, a modified and shared indicator bit, and a valid bit for each cached line. This amounts to $(\frac{b+t+2}{8}) \times p \times w \times l$ total bytes of storage required, where b is the size of the sharer bit vector or pointer list in bits.

For the CMP architecture evaluated in this dissertation, a full-map directory ($b = 128$) with 64 kB caches would require $(\frac{128+27+2}{8}) \times 128 \times 8 \times 256 \approx 4.94$MB, 61.3% of the 8 MB aggregate L2 cache capacity. While a directory area overhead of 61.3% is potentially implementable, it is extremely over-provisioned in the common case when many lines are not unique due to data sharing. Per-sharer cache capacities exceeding the modest 64 kB in the evaluated system would lead to even higher degrees of over-provisioning.

The need for and resulting cost of over-provisioning motivates an approach analogous to a *sparse directory* [65] for memory-based coherence schemes, in which the directory capacity is decoupled from the aggregate first-level cache capacity, enabling smaller on-die *directory caches*. Like a sparse directory, a directory cache must invalidate all shared copies of a line when the line's directory entry is evicted.

The performance impact of a directory cache eviction may be even greater than that of a data or instruction cache eviction; a number of invalidation messages proportional to the number of sharers of the evicted line must be sent over the network, causing congestion, and sharers must re-request the line if they access it again. Given this high cost of evictions, and the power and area costs of sufficiently associative directory caches, a mechanism for minimizing evictions by increasing the *effective* capacity of an on-die directory cache is desirable.

7.1.3 Directory Associativity

We observe that the associativity required by sets in a directory cache varies in three ways. First, at a given point in the execution of a program, different directory sets may have working sets of much different sizes, meaning that one set may require a much more associative directory to achieve good performance than another. Second, a given directory set's working set will grow and shrink

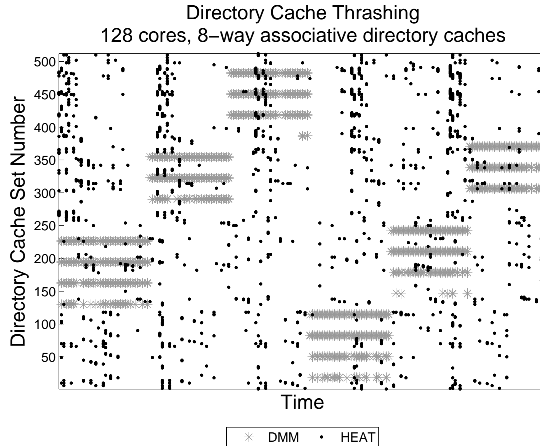


Figure 7.1: Time varying set distribution at last-level (L3) shared cache. The figure demonstrates that directory cache thrashing can vary in time and it may shift across sets, implying that over-provisioning of directory cache entries may be required to avoid performance impact due to directory cache conflicts.

over time. Thus, at different points in time, different directory sets may require high associativity while others are sparsely populated with valid directory entries. Finally, we observe varied associativity profiles across benchmarks, depending on the nature of their computation and communication patterns. These three types of variability imply that a fixed-associativity directory cache cannot adequately address the associativity demands of all directory cache sets across all workloads.

The first type of variability is shown in Figure 7.1. An 8-way set-associative directory cache was simulated for a 128-core (16-sharer) CMP executing a dense matrix multiplication kernel (`dmm`) and a 2D stencil code (`heat`). Figure 7.1 demonstrates that at any point in time, a small number of sets are experiencing the vast majority of evictions. Additionally, this small group of sets is different during different program phases, suggesting that an effective directory scheme must *dynamically* provide high effective associativity to those sets which require it.

The directory cache thrashing effect is influenced by data layout. In our evaluation, we use workloads that have a high degree of optimization applied to them

to avoid these pathologies. These optimizations generally consist of skewing the addresses in memory for segments of data structures with otherwise regular and correlated access patterns across tasks. An example is taking a dense matrix laid out in row-major format and changing its implementation to be an array of pointers. Each pointer points to the start of a row which has been shifted in memory to avoid set conflicts for accesses to the same column in multiple rows. Similarly, the workloads have skewed stacks and heap allocations to avoid conflicts due to local variables and temporary dynamic allocations.

The degree to which conflicts occur at the directory cache banks is also related to the hashing function that maps the address space to DRAM and L3 banks. We have experimented with many permutations of address bits and arrived at one that is sufficient for our workloads. However, even with a good general hash function, there will always be pathological access patterns that lead to set conflicts at the directory cache. Through extensive effort on testing hardware mappings and software modification to reduce conflicts, we have removed all known pathologies from our workloads. We believe our implementation is generally good for most workloads and is robust. However, the existence of pathological cases and the difficulty in finding a generally good mapping function lead us to believe that this is a generally difficult problem that will be an issue for highly banked processors with large numbers of cores.

7.1.4 Replacement Policy

As with data and instruction caches, the replacement policy employed by a directory cache strongly influences the extent to which the cache can capture the working set of an application, along with capacity and associativity. The relationship between high-level cache access frequency and directory cache access

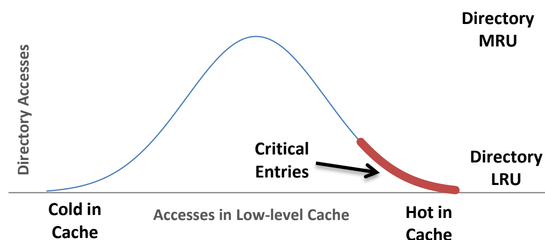


Figure 7.2: General pattern for directory accesses.

frequency is shown in Figure 7.2. In directory coherence schemes, the high-level caches filter the stream of memory references in a program and present a reduced stream at the directory. While a reduced number of accesses at lower levels of the cache and memory is the primary purpose of caches, the filtering reduces the amount of information available for the directory cache to determine line criticality as part of the replacement policy. More concretely, the more frequently accessed, and thus more critical, a line is in a high-level cache, the less likely it is to be evicted from the high-level cache and re-requested from the directory later. Thus, the line will be accessed infrequently at the directory cache and will be a likely candidate for eviction under an LRU replacement policy, as will data infrequently referenced in the program itself.

Given this information gap between first-level caches and directory caches, an LRU replacement policy will incorrectly evict those lines that are most critical to program execution. An MRU policy generally avoids this pathology, but it is still suboptimal because it fails to evict those entries which are truly cold in the first-level cache, and instead evicts entries which are accessed with intermediate frequency. A least-recently allocated (LRA) policy incorrectly evicts long-lived hot data, making it inappropriate as well. The information loss between first-level cache and directory cache makes effective replacement difficult.

Our experiments have shown that lines holding instructions and stack-allocated data from inner-loops of our benchmarks, which are some of the critical entries

in the figure, are particularly pathological because they quickly become LRU in a fixed-sized directory cache while streaming through large volumes of data. WayPoint capitalizes on the infrequency of directory accesses for actively shared lines by removing them from the directory cache, freeing up space for other entries, while not forcing invalidations for performance-critical data nor greatly increasing misses at the directory cache because the actively shared lines remain stable in the directory cache.

7.2 Design

The goal of WayPoint is to approach the performance of a full on-chip directory while requiring much less on-chip storage. To achieve this goal, WayPoint uses set-associative directory caches implemented in hardware with modest associativity. When a directory conflict occurs, WayPoint selects an entry in the cache to evict to a second-level overflow directory resident in cacheable system memory and places the new entry into the directory cache in place of the evicted entry. The remainder of this section describes the design and implementation of the WayPoint first-level on-chip directory cache and second-level overflow directory implemented with linked lists resident in cached memory.

7.2.1 Directory Coherence Protocol

The WayPoint architecture can support various types of directory schemes. We evaluate two common schemes: a full-map directory (Dir_nNB) and a pointer-based limited directory similar to Dir_iB [99]. A Dir_nNB scheme keeps a record of each sharer of a line in a central location and only sends invalidation messages to the *exact* set of sharers of a line when a read-to-write transition or eviction occurs. Tracking the exact set of sharers results in the minimum number of invalidation

messages being sent on a state transition.

A limited directory scheme requires maintaining a pointer to a single write sharer or up to a fixed number of read sharers, as in O'Krafka et al. [66] or Gupta et al. [65]. If the list of pointers overflows, the limited scheme reverts to keeping a count of sharers. After overflowing, at a state transition, the scheme broadcasts an invalidation message to all L2 caches and waits to receive the number of acknowledgements indicated by the counter.

In our evaluation, a set-associative directory cache is used to keep directory entries on-die for both schemes. The directory protocol used is similar to the exclusive-shared-invalid protocol as presented by Hennessy and Patterson [100]. Alternate directory schemes, such as coarse vector [65] or LimitLESS [70], would change the storage efficiency of on-die directory caches by trading off a reduced number of messages sent for a greater number of bits tracked. However, the issue of storage efficiency is distinct from that of directory cache associativity and capacity. Regardless of the composition of directory entries, all directory schemes use a structure to map addresses for cached data to directory entries. Our mechanism provides a way to reduce the capacity and associativity of the mapping structure independent of directory entry composition.

7.2.2 WayPoint Design

The WayPoint architecture is composed of a first-level set-associative on-die cache of directory entries, a collection of head pointers to linked lists holding overflow entries, and WayPoint records and descriptors resident in memory. The architecture also includes top-of-heap and free list pointers for controlling the allocation of overflow entries in memory. A block diagram of the architecture is given in Figure 7.3.

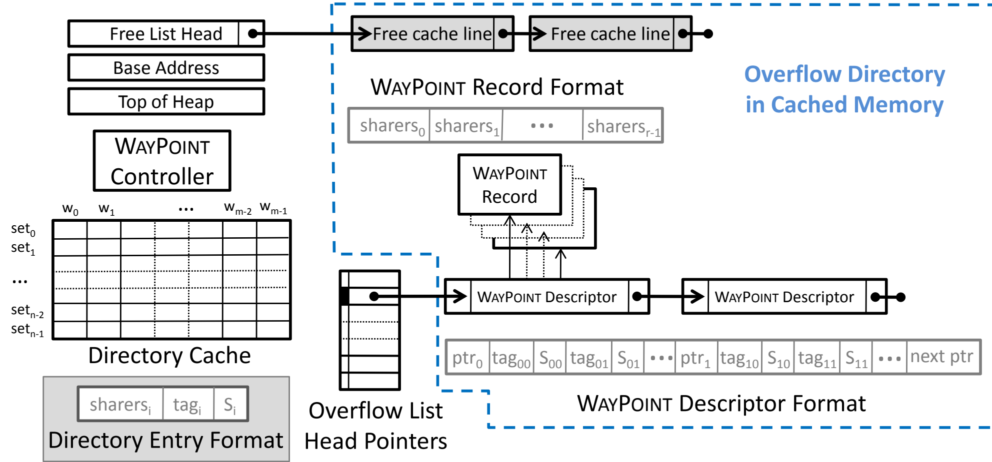


Figure 7.3: Architecture of WayPoint.

A linked list is maintained for each set in the hardware directory cache, forming something akin to a hash table. When an entry must be evicted from the directory cache, it is inserted into the linked list for the set. Each linked list is accessed by dereferencing the appropriate head pointer. Empty linked lists are indicated using a valid bit. Each linked list is composed of WayPoint *descriptors*, which hold the tag and state from some number of directory entries, a set of pointers to WayPoint *records*, which hold sharer state, and a pointer to the next WayPoint descriptor in the list. The metadata for the entries is separated from the sharing list to allow the linked lists to be searched more quickly by accessing the fewest possible number of data elements in memory.

To reduce design complexity, each WayPoint record and descriptor is sized to fit on a single cache line, which is 32 bytes in this dissertation. WayPoint descriptors and records are addressed as an offset from the `base_address` register. The `base_address` register is set to the base physical address of the memory region allocated to WayPoint. The offsets are calculated such that the records and descriptors for a given directory cache are resident in the address space allocated to that directory cache. Mapping directory entries to their respective last-level cache

banks allows memory accesses related to WayPoint to occur locally, without the design complexity and latency of injecting them into the network.

For a full-map directory, we can place two sharer lists into each WayPoint record and enough pointers and associated tag and state information for up to four records in each WayPoint descriptor. For the limited directory scheme with four pointers per entry, we can place up to eight directory entries in each record and need two WayPoint descriptors to hold the associated metadata.

7.2.3 WayPoint Operation

When an entry is not present in the directory cache, the WayPoint controller must respond to events generated by the baseline directory controller logic. The following sections describe each event and how the WayPoint controller responds to the directory controller requests.

7.2.4 Accessing Entries in the Overflow Directory

When a coherence request misses in the directory cache, WayPoint indexes into the table of head pointers. If the valid bit is cleared, i.e., there are no overflow entries for the set, the line is currently not tracked and the directory allocates an entry in the directory cache for the line. We consider this situation a cold cache miss at the directory. If the directory cache has high enough associativity, this becomes the common case for directory cache misses. As a latency optimization, the directory cache and the list of head pointers can be queried in parallel.

When the list is non-empty, the controller traverses the linked list of WayPoint descriptors, comparing the tag of the request to each tag in the descriptors. The controller also checks the state of each entry, comparing against shared and modified entries while skipping invalid entries. If the tag of a valid entry matches

the request, the pointer to the WayPoint record is dereferenced to obtain the sharer list of the directory entry. The directory entry is then reconstructed from the metadata in the descriptor and the sharer list in the record and is provided to the directory cache.

7.2.5 Release Messages

The L2 caches use release messages to notify the directory cache that they no longer hold a line due to an eviction or explicit flush. When a request to the overflow directory is a read-release or a write-release, we could choose to traverse the overflow list for the set to retrieve the directory entry, allocate the entry in the directory cache, and perform the update there, or simply update the overflow entry in-place without modifying the contents of the directory cache. Since most releases are due to capacity-related evictions at the L2 caches, many of these releases are to lines that will not be accessed for some time. The lack of temporal locality would make filling the directory entry into the directory cache a low-utility operation since it is likely to get evicted to the overflow list; therefore, we do not fill on a release. Moreover, releases do not block the cores. The release messages are processed by the L2 cache controller which operates concurrently with the first-level cache of the core. The cores in a cluster can continue to issue requests into the network while releases are occurring, and thus low latency is not critical.

For write releases, or the last read-release to a shared line, the WayPoint entry holding the line can simply be invalidated since the only sharer has now relinquished ownership. When no valid WayPoint data is in the line that was holding the entry or descriptor, those lines are unlinked and returned to the free list.

7.2.6 Overflow Directory Removal

Read or write requests that hit in the directory cache proceed as normal. A miss in the directory cache will traverse the overflow list for the set, if valid, and fill into the directory cache. If the entry is not found in the overflow list or the directory cache, a new entry is allocated. In the case of requests that hit in the overflow list and thus must replace an entry in the directory cache, eviction is simplified since the LRU entry in the directory cache can be swapped with the record. If a free directory cache entry exists, the swap is obviated and the controller fills the request into the free entry and invalidates the existing entry in the overflow list. Once the directory entry for the request is in the cache, the directory access proceeds as normal.

For performance reasons, it may be necessary to have a mismatch between the number of overflow lists and the number of sets to reduce the average length of a list or to minimize the number of head pointers. When the number of overflow head pointers does not match the number of sets, a swap cannot always occur. A swap can fail to work when the list holding the requesting entry does not match the list that the entry being evicted from the directory cache must be inserted into. In such cases, WayPoint performs a removal and an insertion using two separate lists. The additional traversals result in additional overhead; but in directory caches with a small number of sets, the performance gained by having more overflow lists far outweighs the cost added to swap operations.

7.2.7 Hardware Directory Cache Eviction

In the case where a swap cannot be used to evict an entry from the directory cache, the WayPoint controller must traverse the linked list of overflow entries, find or allocate a free record and descriptor, and fill the evicted entry into those

lines resident in memory. An example of when a swap would not occur is when a line is accessed for the first time at the directory, and thus needs to fill into the directory cache, but all directory entries in the set that the access maps to are valid. Under these conditions, an eviction is necessary and there is no swap candidate.

Note that entries in the directory cache that are in a transient state are not allowed to be evicted. An example of a transient state is an entry that is stalled waiting to collect invalidation requests while transitioning from the shared state to modified or exclusive state. The eviction can stall while the controller waits for responses to return. When the responses are collected, the target of the eviction is moved into a non-transient state. If transient requests were allowed to be evicted to the overflow directory, thrashing could occur.

When an allocation in the overflow directory is required, the `free_list` register is checked. If it is non-NULL, the head of the free list is swapped with its `next` pointer, and the allocated line is linked into the overflow list as a descriptor or into a descriptor as a new WayPoint record. Deallocation of a cleared descriptor or record is performed by setting the `next` pointer of the free block to the value in `free_list` and then overwriting `free_list` with the address of the free block. Note that operations performed to the free list are LIFO to increase the probability of a cache hit when the list is accessed. Since the values in lines in the free list are irrelevant, write-to-own semantics at the L3 for WayPoint could also reduce the latency of accessing lines in the free list by avoiding unnecessary off-chip memory accesses.

If the `free_list` pointer is NULL, a new line is allocated by incrementing the `top_of_heap` register to generate an address that maps into the address region cached by the cache bank associated with the directory. The heap is only used to allocate lines since allocated lines are either used by the overflow directory or

the lines are in the free list. We allocate the maximum space required for all possible entries expected, i.e., there will never be an out-of-memory condition. It is possible to handle out-of-memory conditions by invalidating current overflow entries, but for simplicity we do not investigate such measures in this dissertation. Fragmentation may also be possible in overflow lists if a large number of records are allocated and then freed, but not cleared and deallocated. Compaction could be performed periodically to address this issue, but fragmentation was not found to be an issue in our implementation.

7.3 Summary and Discussion

In the design, implementation, and evaluation of WayPoint, we observe potential bottlenecks and optimization opportunities to be addressed in future work. Ordering of overflow entries in the overflow directory, replacement policy for the directory cache, and the extra time required to determine if a directory cache miss is an overflow directory hit are potential bottlenecks. While added overhead due to ordering of overflow lists and traversals for determining residency in the overflow list would appear to be performance-limiting, they were not found to be a first-order concern in our initial implementation. LRU replacement in the directory cache was also suitable for WayPoint; however, we found it to be a poor choice for a fixed directory without WayPoint due to the filtration effect of lower-level caches with regard to directory cache criticality information.

7.3.1 Optimizations

The length of overflow lists could be reduced by increasing the number of lists in the WayPoint design. Such a technique trades off additional area to implement more head pointer registers for faster searches due to reduced list length. A second

potential mechanism is a counting Bloom filter [101] that uses hashes of directory entry addresses to increment counters when an entry is inserted into the overflow list. Finally, the order of entries in the lists could be modified dynamically to put frequently accessed entries near the beginning of the list to reduce traversal time.

Incomplete knowledge of memory access patterns limits the performance of directory cache replacement policies. Propagating LRU status from low-level caches to the directory using a side channel could mitigate the problem of LRU invalidating directory cache entries that are often accessed by first-level caches but are seldom accessed at the directory. However, WayPoint already deals with such cases by avoiding invalidation, and instead WayPoint uses overflow lists making replacement policy much less of a concern.

The WayPoint controller must access memory where the overflow lists are stored. In doing so, WayPoint must contend with normal requests for access to the top-level caches and the memory controller. In practice, the impact of contention is low since WayPoint must only access the data cache on a miss in the directory cache, which is the infrequent case, and many of those accesses hit in the LLC, minimizing the memory bandwidth requirements of the directory.

7.3.2 Impact of Other Workloads

In this dissertation we focus on data- and task-parallel workloads similar to those investigated in previous work for accelerators and CMPs with 64+ cores [12,15,98]. We note that other applications may change the rate and nature of directory requests and would thus impact the performance of the design described here. However, the parameters of the WayPoint design can be changed to re-target different workloads by increasing the allowed number of outstanding requests or the number of directory banks in the design. Moreover, the interaction between

number of invalidation messages sent and choice of limited versus full-map directory scheme is strongly influenced by the sharing patterns of the workloads, but the choice of scheme to implement is independent of WayPoint. We leave the investigation of other workloads and the comparison of limited, full-map, and other directory schemes to future work.

7.4 Summary

In this chapter we present WayPoint, a lightweight hardware mechanism for virtualizing the directory cache size by placing evicted directory entries into linked lists in cached memory. The result is a smooth degradation in performance when the on-die working set exceeds the on-die directory cache capacity. This smoothing enables the use of smaller on-die directory caches, which in turn reduces power and area overhead of coherence management. The advantage of WayPoint is its ability to implement cached directories that cover a large fraction of LLC requests while making the small fraction that WayPoint does not capture in its on-die cache only marginally more expensive to access.

WayPoint provides tolerance for unbalanced set and bank accesses at the directory and reduces the need to over-provision directory cache resources. WayPoint allows for both inclusive and exclusive last-level cache implementations and supports small, low-associative directory caches.

CHAPTER 8

Cohesion

The main contribution of this dissertation is the exploration and evaluation of a hybrid memory model for multicore processors with 1024+ cores. The defining characteristic of a system supporting a hybrid memory model is the existence of a method by which responsibility of coherence management can be transferred between software and hardware. Cohesion is a protocol that enables the necessary transitions between coherence domains at runtime. Cohesion works at the granularity of a cache line. It relies upon the existence of a software-managed coherence protocol, such as the Task-Centric Memory Model presented in Chapter 5, and an underlying hardware coherence mechanism, such as the SPF or WayPoint presented in Chapters 6 and 7, respectively, to expose a single coherent view of memory to software without full hardware coherence support. With small additions to the hardware, Cohesion allows software to move cache lines in and out of the hardware coherence protocol.

8.1 Motivation

We motivate Cohesion by demonstrating that both software-managed and hardware-managed coherence have overheads that can be mitigated by a hybrid memory model. We also motivate Cohesion from a programmability and optimization perspective. Having hardware cache coherence available, even if a high performance penalty is paid for its use, can enable an optimization path via iterative

Table 8.1: Differences in design goals and architectural features between general-purpose CPUs and accelerators such as GPUs.

	Conventional Multicore	Accelerators
Examples	Intel i7, Sun Niagara, AMD Opteron	Cell, NVIDIA GPUs, ATI GPUs
Optimized for	• Minimal latency • Tightly coupled sharing • Fine-grained synchronization • Coherence transparent to programmer	• Maximal throughput • Loosely coupled sharing • Coarse-grained synchronization • Special-purpose functionality
Architecture supports	• Single address space • Hardware caching • Strict consistency models • Hardware cache coherence	• Multiple address spaces • Software-managed scratchpads • Relaxed consistency models • Software-managed coherence

refinement. Developers and tools can migrate data into software-managed coherence when possible and profitable, but such remapping is not required for correctness.

We find a compelling use case in heterogeneous processors with general-purpose and accelerator cores with a shared single address space. In such a design, where the coherence needs and capabilities vary by type of core, a hybrid approach may be beneficial by reducing the need for data marshalling and the cost of data copies. Table 8.1 lists the differences in the design goals and the features present in those architectures. We would like to build a system tuned to the applications that are scalable while supporting a more conventional programming model. To do that, we borrow the design goals of accelerators from the table and use Cohesion to support the architecture features of more conventional general-purpose platforms. The end result is a system with CPU-like programmability with accelerator-like scalability and performance.

8.1.1 The Case for Software-Managed Cache Coherence

A software-managed coherence protocol embedded in the compiler [74, 93], runtime [102, 103], or programming model [71, 88] avoids the overheads of hardware

coherence management. No memory is required for directories [67, 99] or duplicate tags [2, 64]. No design effort is expended implementing and verifying the coherence protocol. There is potential for reduced network traffic and relaxed design constraints. Furthermore, software protocols can mitigate or eliminate false sharing.

Software-managed cache coherence (SW_{cc}) has the ability to mass invalidate shared data, signaling many invalidations with only a few messages. To coordinate this action, a global synchronization event such as a barrier is used. As shown in Chapter 7, the equivalent operation in hardware requires sending potentially many invalidation messages exactly when a state transition or eviction is needed. Such an invalidation mechanism lengthens the critical path for coherence actions, increases contention in the network, and requires greater capacity from the network to handle the invalidation traffic. SW_{cc} can eliminate false sharing since multiple write sharers that access disjoint sets of words on a line will not generate coherence probes that would otherwise cause the line to ping-pong between sharers in hardware cache coherence.

8.1.2 The Case for Hardware Cache Coherence

Hardware cache coherence (HW_{cc}) provides a number of programmability benefits. HW_{cc} can enforce strict memory models whereby hardware ensures that all reads receive the latest write to a word in memory, which makes it easier to reason about sharing in some applications. HW_{cc} enables speculative prefetching and data migration. Shared memory applications can be ported to a HW_{cc} design without a full rewrite, albeit with possibly degraded performance. While it is unlikely that shared memory applications targeting contemporary multicore processors with four to eight cores will trivially scale onto 1000-core CMPs, it is

possible that a developer can decompose the parallelism such that the same code will run reasonably well on past, present, and future systems spanning one to perhaps 32 or 64 cores without a rewrite. However, this projection is predicated on maintaining the same ISA and memory model across generations of the processor. Regardless of performance scaling, if a 1000-core CMP supports the same ISA and coherent memory model as that of a contemporary processor, it is reasonable to assume the same code written for today's multicore will at least run *correctly* on future 1000-core CMPs. The situation is akin to sequential applications written before the advent of multicore processors that are able to continue running on contemporary CMPs, which maintain a backwards compatible ISA and memory model. Cross-generational parallel application support may be infeasible if the coherence mechanisms are built into the application and are dependent on a fixed microarchitecture.

Another way to characterize the difference between SW_{cc} and HW_{cc} is by observing the semantics of their communication. SW_{cc} is a *push* mechanism, and thus explicit actions must occur to make data modified by one sharer visible to other sharers. HW_{cc}, though, is a *pull* mechanism, allowing a requester to locate the latest copy of the data on-demand. The implication is that SW_{cc} protocols may be more conservative than necessary, pushing all data that may be read by another core to a globally visible point, e.g., memory or the globally shared last-level cache. Hardware coherence protocols do not constrain software from a correctness perspective since they rely on hardware to track or discover coherence state as needed. Furthermore, while the additional traffic required for read release messages under HW_{cc} makes up a significant portion of the message traffic, these messages are not on the critical path for a waiting access as is an invalidation sent by the directory. SW_{cc} pushes all data out of the cache that *may* be read, while HW_{cc}, ignoring the impact of prefetching, only moves data that are needed,

which results in greatly reduced traffic and cache misses for a class of workloads.

8.1.3 A Hybrid Memory Model

Supporting multiple coherence implementations enables software to dynamically select the appropriate mechanism for blocks of memory. Supporting incoherent regions of memory allows more scalable hardware by reducing the number of shared lines, resulting in fewer coherence messages and less directory overhead for tracking sharer state. Furthermore, having coherence as an option enables trade-offs to be made regarding software design complexity and performance. Even for applications that do not need data to transition frequently between SW_{cc} and HW_{cc}, a hybrid memory model provides the runtime with a mechanism for managing coherence needs across applications. Put another way, hardware cache coherence allows the runtime or operating system to put memory into a consistent state even when software performs an incorrect action.

From the perspective of system software, HW_{cc} has many benefits. HW_{cc} allows for migratory data patterns not easily supported under SW_{cc}. As discussed in Chapter 3, the implementation of Rigel's runtime system, the Rigel Task Model, required coherence. The lack of hardware coherence for all memory led to the use of global memory operations, which are not locally cacheable, resulting in increased bandwidth and latency for RTM memory accesses that are otherwise amenable to caching.

Threads that sleep on one core and resume execution on another would need to have their local modified stack data available, forcing coherence actions at each thread swap under SW_{cc}. Likewise, task-based programming models [79, 80] are aided by coherence. HW_{cc} allows children tasks to be scheduled on the same core as their parent, incurring no coherence overhead, or stolen by another core which

would allow data to be pulled using HW_{cc}.

Systems-on-a-chip, which incorporate accelerators and general-purpose cores, are available commercially [104] and are a topic of current research [16]. The assortment of cores makes supporting multiple memory models on the same chip attractive. A hybrid approach allows for cores without HW_{cc} support, such as accelerator cores, to cooperate with cores that do have HW_{cc} support and interface with coherent general-purpose cores. While a single address space is not a requirement for heterogeneous systems, as demonstrated by the Cell processor [57] and GPUs [4], it may aid in portability and programmability by extending the current shared memory model to future heterogeneous systems. A hybrid approach allows for HW_{cc} to be leveraged for easier application porting from conventional shared memory machines and easier debugging for new applications. SW_{cc} could then be used to reduce the stress on the hardware coherence mechanisms to improve performance.

8.1.4 Summary

Hardware-managed and software-managed cache coherence offer both advantages and disadvantages for applications and system software. We list many of the trade-offs in Table 8.2. A hybrid memory model such as COHESION leverages the benefits of each while mitigating the negative effects of the respective models. The key benefits from SW_{cc} are reduced network and directory costs and the potential to avoid false sharing without programmer intervention. The key benefits from HW_{cc} are its ability to share data without explicit software actions, which, as we demonstrate, can be costly in terms of message overhead and instruction stream inefficiency. A hybrid approach can enable scalable hardware-managed coherence by supporting HW_{cc} for the regions of memory that require it using SW_{cc} for data

Table 8.2: Trade-offs for HW_{cc}, SW_{cc}, and COHESION.

	Programmability	**Network Constraints**	**On-die Storage**
HW_{cc}	Conventional CMP shared-memory paradigm; supports fine-grained, irregular sharing without relying on compiler or programmer for correctness	Potential dependences handled by hardware instead of extra instructions and coherence traffic	Optimized for HW_{cc}: when HW_{cc} desired, coherence data stored efficiently
SW_{cc}	Used in accelerators; provides programmer/compiler control over sharing	Eliminates probes/broadcasts for independent data, e.g., stack, private, immutable data	Optimized for SW_{cc}: minimal hardware overhead beyond hardware-managed caches
COHESION	Supports HW_{cc} and SW_{cc}; clear performance optimization strategies allowing $SW_{cc} \Leftrightarrow HW_{cc}$ transitions	SW_{cc} used to eliminate traffic for coarse-grain/regular sharing patterns; HW_{cc} for unpredictable dependences	Reduces pressure on HW_{cc} structures; enables hardware design optimizations based on HW_{cc} and SW_{cc} needs

that does not. In comparison to a software-only approach, a hybrid memory model makes coherence management an optimization opportunity and not a correctness burden.

8.2 Design

COHESION provides hardware support and a protocol to allow data to migrate between coherence domains at runtime, with fine granularity, and without the need for copy operations. Figure 8.1 shows the relationship between the HW_{cc} and SW_{cc} protocols. The default behavior for COHESION is to keep all of memory coherent in the HW_{cc} domain. Software can alter the default behavior by modifying tables in memory that control how the system enforces coherence. Data that are not shared, or that can have coherence handled at a coarse granularity by software, use the SW_{cc} domain and no hardware coherence management is applied.

The rest of this section describes the protocols and hardware support required for COHESION that enable on-the-fly coherence domain changes. Note that with minor restrictions, the selected hardware and software protocols used by COHESION could be exchanged for other implementations, but the basic technique

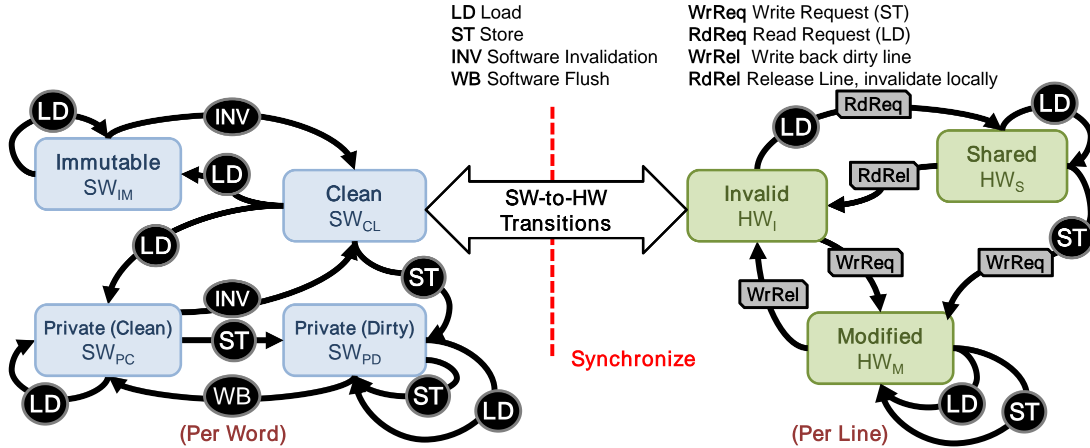


Figure 8.1: COHESION state diagram.

provided by this work would remain the same.

8.2.1 Hardware Coherence Protocol

We now describe the hardware coherence protocol that we use to demonstrate COHESION. These choices are not fundamental to COHESION and could be altered without changing its underlying benefits. Data in the HW_{cc} domain is tracked by an on-die full-map directory [67] implementing an MSI protocol. An exclusive state is not used due to the high cost of exclusive to shared downgrades for read-shared data. Owned state is omitted since we use the L3 to communicate data and the directory to serialize accesses, removing much of the benefit of sourcing shared data from another L2.

While developing the protocol we analyze the benefit of sourcing data from another L2 and found the benefit to be minimal for most workloads. Most L2 misses that could be serviced by another L2 were serviced by the L3. For those L2 misses that also generated L3 misses, but could be serviced by another L2, the added latency of going to memory versus accessing another L2 was not sufficient to make any benefit of L2-to-L2 transfers significant.

For overall runtime experiments, a limited directory scheme [99] (Dir_4B) is used due to its lower storage overhead. For limits studies, we use a full-map scheme, which we use to provide a lower bound on the performance impact and message overhead of pure hardware coherence. We employ sparse directories [65,66] where the directory only holds directory entries for lines present in at least one L2. For evaluating Cohesion we avoid a full directory due to the resulting high memory overhead. Note that the storage overhead is worse for CMPs than multi-socket systems since the overhead grows linearly in the number of sharers, i.e., cores, and not sockets. Therefore, memory bandwidth does not scale with directory size as in multi-socket multiprocessors with directories.

Duplicate tags [64] were not chosen due to their high required associativity (2048 ways) and the difficulty of supporting a multi-banked last-level cache, which may require replicating the duplicate tags across L3 banks. The difficulty with duplicate tags arises when trying to develop a mapping from addresses to sets in the L2 and banks in the L3. If a set in the L2 can hold lines from different banks of the L3, the duplicate tags at the L3 must be over-provisioned by a factor proportional to the number of different L3 banks that may be present in each L2 set. More generally, duplicate tag schemes add constraints to the design of the cache hierarchy. The design of the duplicate tag scheme in [105] illustrates the L1-to-L2 mapping constraints the designers used to make duplicate tags tractable. In such a design, the last-level cache, which is the L2 in [105] and the L3 in our design, must be inclusive, which may be a poor choice for a system with a large number of cores and thus a high aggregate private cache capacity. Moreover, duplicate tags require high associativity and thus require structures that are difficult to implement in CMOS.

The baseline architecture is non-inclusive between L2 and L3 caches. The directory is inclusive of the L2s and thus may contain entries for lines not in the

L3 cache. L2 evictions notify the directory. If the sharer count drops to zero, the entry is deallocated from the directory. Entries evicted from the directory have all sharers invalidated. We could choose to invalidate lazily, relying on broadcasts when a directory miss occurs and thus saving the cost of issuing release messages from the L2 on a clean eviction. However, doing so increases the cost of cold cache misses since all L2s must be probed before a directory allocation can occur. Moreover, the scalable probe filter is an extension of that idea evaluated in Chapter 6 and was found not to scale to 1024 cores for many workloads.

One bank of the directory is attached to each L3 cache bank. All directory requests are serialized through a home directory bank, thus avoiding many of the potential races in three-party directory protocols [106]. Associating each L3 bank with a slice of the directory allows the two mechanisms to be co-located, reducing the complexity of the protocol implementation compared to a design where a directory access may initiate a request to a third-party structure, such as another L2 cache as in Origin [106] or an L3 bank that is across the network.

8.2.2 Software Coherence Protocol

Our software coherence protocol is a variant of the Task-Centric Memory Model [88] adapted to our platform to support hybrid coherence, as shown on the left side of Figure 8.1. The protocol leverages the bulk-synchronous [60] (BSP) compute pattern. BSP comprises phases of mostly data-parallel execution followed by communication, with barriers separating each phase. The software protocol makes use of the fact that most data is not read-to-write shared across tasks between two barriers and most inter-task communication occurs across barriers. Other protocols for managing coherence in software could be used, but we restrict ourselves to a model based on BSP for simplicity of illustration and its broad applicability

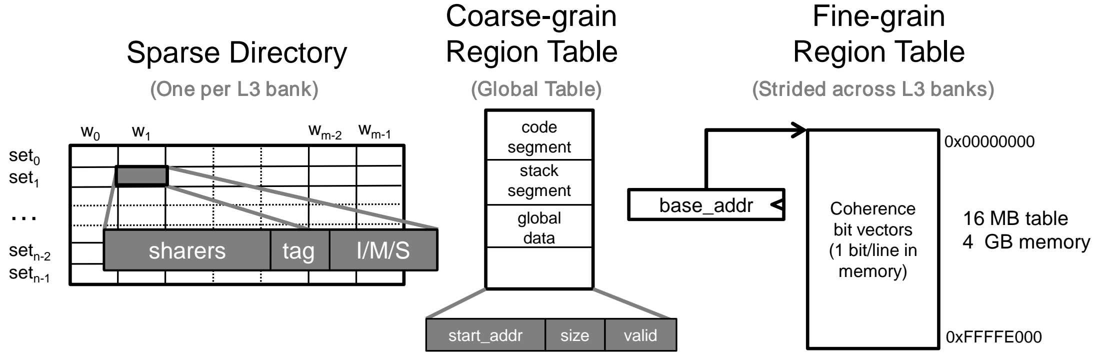


Figure 8.2: COHESION architecture.

to existing scalable programming models.

The software protocol provides a set of state transitions, which are initiated explicitly by software or implicitly by hardware, that allow for a programmer or compiler to reason about coherence for a block of data. The motivation for designing such a protocol is that in a system with caches, not scratchpads, and without coherence there is the potential for hardware to implicitly move data into a globally visible location. These implicit actions are uncontrollable by software, and thus the SW_{cc} protocol, and by extension COHESION, must take them into account.

8.2.3 COHESION: A Hybrid Memory Model

COHESION achieves hybrid coherence by tracking the coherence domain to which regions of memory belong and orchestrating coherence domain transitions. As shown in Figure 8.2, the system is composed of a directory for tracking currently shared HW_{cc} lines, a coarse-grained region table for tracking common large regions of memory that are SW_{cc}, and a fine-grained region table that is used for tracking the rest of memory that may transition between HW_{cc} and SW_{cc}. One bit of state per line, the *incoherent bit*, is added to the L2 cache to track which cached

lines are not HW_{cc}. A compressed hardware structure, such as the structure used in [107], was considered to move all tracking information on-die. However, we find a bitmap cached at the L3 to be a sufficient approach. If additional L3 latency for table accesses becomes a concern, the dense structure of the table is amenable to on-die caching similar to the approach used in WayPoint.

When a request arrives at the L3, the directory is queried. If the line is a directory hit, the line is in HW_{cc} and the response is handled by the directory. If the line is accessible to the requester, the L3 is accessed in the next cycle and the response is returned to the requesting L2. A directory hit with an L3 miss will result in the directory access blocking. A response is generated when the fill from memory occurs or a response from an L2 when the line is in the modified state at the directory. A directory miss results in the region tables being examined.

The coarse-grained region table is a small on-die structure that contains a map of address ranges that are in the SW_{cc} domain. The structure is accessed in parallel with the directory. The three regions used most frequently are for code, private stacks, and persistent globally immutable data. When an access misses in the directory and the address maps into one of these ranges, the L3 cache controller responds with the data. The message includes a bit signalling to the L2 that an incoherent access has occurred. When the response arrives, the L2 sets the incoherent bit in the L2 cache tag for the line. Under SW_{cc}, if the line is invalidated by software or evicted while in the clean state, the line is dropped and no message is sent from the L2 to the L3. While global *visibility* is not required for SW_{cc} data, e.g., private stack data, any modification to such data must be persistent even when such lines overflow the capacity of the L2 cache. When dirty data in the L2 with the incoherent bit set is evicted, it is written back.

For all other accesses, the fine-grained region table is queried. The table may be cached in the L3 since the L3 is outside of the coherence protocol, which only

applies between the L2 caches in our implementation. We map all of memory using one bit per cache line. For a 4 GB address space, a map of all memory would require 16 MB total. To reduce the footprint of the table, a subset of memory could be designated as the COHESION-enabled region. A fine-grained region table lookup must access the L3. A minimum of one cycle of delay is incurred by fine-grained lookups and more if contention at the L3 or an L3 cache miss for the table occurs. If the bit in the table is set, the L3 responds with the data and sends the incoherent bit along with the message. If the bit is cleared, an entry for the corresponding line is placed into the directory. The line is returned to the requester and thereafter it is kept hardware coherent. Should a directory eviction occur that is followed by a future access to the table, the directory entry will be reinserted.

The region tables are set up by the runtime at initialization. The bootstrap core allocates a 16 MB region for the fine-grained region table, zeroes it, and sets a machine specific register to the base address of the table in physical memory. The process is akin to setting up a hardware-walked page table. To toggle the coherence domain of a line, the runtime uses global atomic instructions, `atom.or` and `atom.and`, that bypass local caches and perform bitwise operations at the L3 to set or clear bits in the table, respectively. Atomic read-modify-write operations are necessary to avoid races between concurrent updates to bits within a word of the table.

To make COHESION microarchitecturally agnostic, we must provide special consideration for how we calculate an address to modify an entry in the table. The table is distributed across the L3 banks in our design. To remove the need for one L3 bank to query another L3 bank on a table lookup, we map the slice of the table covering one L3 bank into the same L3 bank it maps to. Since the address space strides across L3 banks, the target address that we want to update

Table 8.3: Programmer-visible software API for COHESION.

API Call	Description
`void * malloc(size_t sz)`	Allocate memory on coherent heap. Data is always in HW_{cc} domain. Standard libc implementation.
`void free(void * hwccptr)`	Deallocate object pointed to by `hwccptr`. Standard libc implementation.
`void * coh_malloc(size_t sz)`	Allocate memory on the incoherent heap. Data is allowed to transition coherence domains. Initial state is SW_{cc} and the data is not present in any private cache.
`void coh_free(void * swccptr)`	Deallocate object pointed to by `ptr`.
`void coh_SWcc_region(void * ptr, size_t sz)`	Make region `ptr` part of the SW_{cc} domain. Data may be HW_{cc} or SW_{cc} data.
`void coh_HWcc_region(void * ptr, size_t sz)`	Make region `ptr` part of the HW_{cc} domain. Data may be HW_{cc} or SW_{cc} data.

in the table must be hashed before being added to the table base address[1]. Since the hash function is dependent upon the number of L3 banks, we choose to add an instruction to perform the hashing. The `hybrid.tbloff` instruction takes the target address and produces a word offset into the table that can be added to the base address before accessing the table from hardware. The instruction makes COHESION microarchitecture-agnostic since the number and size of L3 banks and the stride pattern can be changed without modifying software.

8.2.4 Software Interface to COHESION

In this section we discuss the application programming interface (API) to COHESION. The API calls are listed in Table 8.3. For this work we make two simplifying assumptions. First, we assume there is a single application running on the system. Second, we assume a single 32-bit address space where physical addresses match virtual addresses. The architecture we propose could be virtualized to support multiple applications and address spaces concurrently by using per-process region tables. However, the full details of such an implementation are outside the scope of this work.

[1]DRAM row stride is used. We use $addr_{[10..0]}$ map to the same memory controller and $addr_{[13..11]}$ are used to stride across controllers. The hashing function for an eight-controller configuration would use $addr_{[9..5]}$ to index into the word, and table word offset address would be $addr_{[31..24]} \circ addr_{[13..11]} \circ addr_{[23..14]} \circ addr_{[10]} << 2$.

The COHESION region tables are initialized by the runtime when the application is loaded. The coarse-grained SW_{cc} regions are set for the code segment, the constant data region, and the per-core stack region. The code segment and constant data address ranges are found in the ELF header for the application binary. Our architecture does not support self-modifying code so HW_{cc} is not required for cached instructions.

The stack address range ends at the top of memory and starts at the base of the per-core stack region, whose size is nominally the number of hardware threads times the initial stack size per thread. Variable-sized stacks are possible by treating the stack region as a SW_{cc} heap, but fixed-sized stacks were found to be sufficient for our current implementation.

There are two heaps in our implementation: a conventional C-style heap that is kept coherent, and another that is not kept HW_{cc} by default. The *incoherent heap* is used for data that may transition coherence domains during execution. Note that the minimum-sized allocation on the incoherent heap is 64 bytes, or two cache lines, so that the metadata for the allocation can be kept coherent. We ran an experiment on Ubuntu Linux 9.10 running glibc 2.10 to determine minimum heap allocation sizes. Our evaluation confirmed that current libc implementations require 16 to 32 byte minimum allocations, so we believe 64 bytes to be reasonable.

8.2.5 Coherence Domain Transitions

A transition between SW_{cc} and HW_{cc} is initiated by word-aligned, uncached read-modify-write operations performed by the runtime to the fine-grained region table. The issuing core blocks until the transition is completed by the directory for the purposes of memory ordering. Blocking allows for software to enforce an order between accesses before and after the table modification. To implement a synchro-

nization construct at the point of table modification, the modifying core would need to globally synchronize, perform the transition operation, and synchronize again to notify the cores that the new state is in effect. While this may seem heavy handed, updates can be done in-bulk by making use of programmatic barriers, such as those used in RTM, thus amortizing the synchronization overhead.

There are minor aspects of the hardware coherence protocol implementation that are necessary for domain transitions to occur correctly. All coherence actions initiated by the L2, such as read releases and writeback operations, are acknowledged by the directory to ensure there are no lost updates occur while coherence domain transitions are in progress. Acknowledging coherence messages also reduces the complexity of the protocol and allows for reordering in the network, both of which were design objectives for our implementation. Writebacks are always acknowledged, even when the line being written back is not kept coherent by hardware, to avoid lost updates due to domain transitions and writebacks racing in the network. Even if ordering between the L2 and directory were disallowed, messages of different classes could pass one another in the network leading to deadlock or lost updates. We also use no timeouts in our protocol or negative acknowledgments that require retransmission on the part of the sender. Other implementations and more complicated protocols could be implemented and shown correct, but those are outside the scope of this work.

The runtime can transition SW_{cc} (HW_{cc}) lines to be HW_{cc} (SW_{cc}) by clearing (setting) the corresponding state bits in the fine-grained region table. If a request for multiple line state transitions occurs, the directory serializes the requests line-by-line. State transition requests require a single MSHR at the L2 regardless of the number of lines to be modified by the request. We pack coherence domain state information densely into a contiguous region of memory with one bit of state per tracked line denoting HW_{cc} or SW_{cc}. In this way, up to 256 state transitions

are possible given that we use a line size of 32 bytes. All lines that may transition between coherence domains are initially allocated using the incoherent heap in our implementation, and the initial state of these lines is SW_{cc}.

The directory controller is responsible for orchestrating the $HW_{cc} \Leftrightarrow SW_{cc}$ transitions. The directory snoops the address range for the fine-grained region table and upon an access that changes the coherence domain of a line, the directory performs the actions described below to transition between SW_{cc} and HW_{cc}. The request completes by sending an acknowledgment to the issuing core. Handling the transitions at the directory allows for requests for a line to be serialized across the system. It also imposes order between domain transitions and normal memory accesses that arrive at the directory and L3. Coherence domain transitions for a single line thus occur in a total order across the system with all other coherent and non-coherent accesses at the L3 being partially ordered by the transitions.

8.2.6 $HW_{cc} \Rightarrow SW_{cc}$ Transitions

To move a line out of the hardware coherent domain requires removing any directory state associated with the line, updating the table, and putting the line in a consistent state known to software. Figure 8.3 shows the potential states a line can be in when software initiates a $HW_{cc} \Rightarrow SW_{cc}$ transition. In our examples, we only show two L2 caches for simplicity. Each of the states corresponds to a possible state allowed by the MSI directory protocol. After a transition is complete, the line is not present in any L2 and the current value is present in the L3 or memory.

For Case 1a of Figure 8.3, the directory controller queries the directory and finds the entry not present, indicating that there are no sharers. Therefore, no action must be taken other than to set the bit in the region table. Case 2a has the

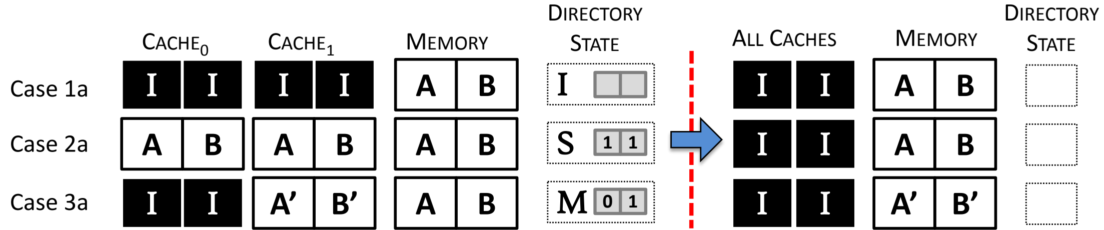


Figure 8.3: Cache state transitions between HW_{cc} and SW_{cc}.

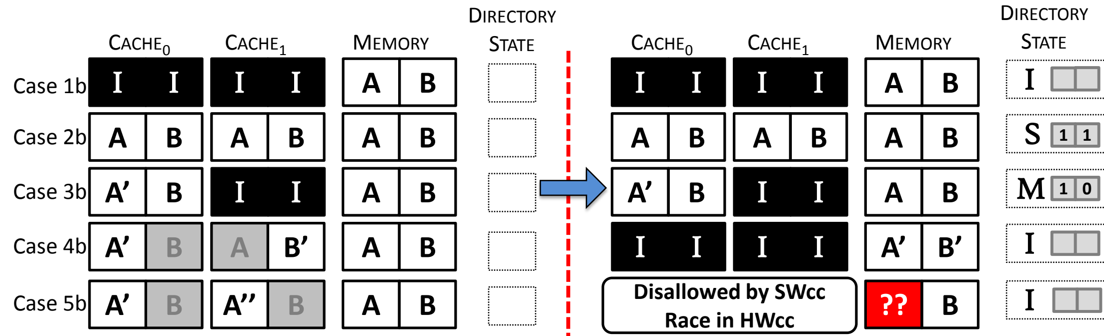


Figure 8.4: Cache state transitions between SW_{cc} and HW_{cc}.

line in the shared state with one or more sharers. The directory performs a directory eviction that invalidates all sharers of the line. When all acknowledgments are received, the directory entry is removed, the bit is set in the region table, and the response is returned to the core requesting the transition. When a line is in the modified state, shown in Case 3a, a newer version of the data exists in some L2. The directory sends a writeback request to the owner. When the response arrives, the L3 is updated, the table is modified, and a response is sent to the requester to unblock it.

8.2.7 $SW_{cc} \Rightarrow HW_{cc}$ Transitions

The left half of Figure 8.4 shows the potential states that a line may be in for two L2 caches when a line is under SW_{cc}. Since the directory has no knowledge of the line state, a transition to HW_{cc} initiates a broadcast clean request sent from the directory. When the line is found in an L2 in the clean state, the incoherent bit is

cleared, i.e., the line is now susceptible to cache probes, but the line is not evicted from the L2. Clean lines send an acknowledgment message to the directory, which adds the L2 to the list of sharers. If the line is not found in the L2, a negative acknowledgment is sent to the directory. Figure 8.4, Cases 1b and 2b demonstrate transitions for lines that are not modified.

If a dirty line is found, shown in Cases 3b and 4b, the L2 sends the directory a notification. If there are any read sharers, the directory sends messages forcing all readers to invalidate the line and the owner to writeback the dirty copy. If the line is dirty in only one cache, the sharer is upgraded to owner at the directory and no writeback occurs, saving bandwidth.

If multiple writers are found, writeback requests are sent to all L2s holding modified lines and invalidations are sent to all L2s holding clean lines. Our architecture maintains per-word dirty bits with the cache lines allowing the L3 to merge the result of multiple writers if the write sets are disjoint. When either operation is complete, the line is not present in any L2 and the L3 or memory holds the most recent copy of the line.

Note that the system can always force a $SW_{cc} \Rightarrow HW_{cc}$ transition to make caches consistent, such as between task swaps, but the data values may not be safe. It is possible for faulty software to modify the same word of the same line in two L2 caches concurrently when under the control of SW_{cc}(Figure 8.4, Case 5b). This represents a hardware race. To safely clean the state of a word, the runtime can always turn on coherence and then zero the value. It will cause the dirty values in the separate caches to be thrown away and the zeroed value to persist. For debugging, it may be useful to have the directory signal an exception with its return message to the requesting core.

8.3 Protocol Optimizations

In most synchronized programs, one can expect the number of active sharers to be zero after the point of synchronization preceding a write. Likewise, the number of readers after the write, but before synchronization, should also be zero. The fact that most read-to-write sharing occurs across synchronization points was leveraged in past systems, such as the Dir_1SW protocol [71], and is used by our own SW_{cc} protocol, with barriers acting as the synchronization point. The same pattern of sharing can be leveraged to optimize for two common cases we see in $HW_{cc} \Leftrightarrow SW_{cc}$ transitions.

Lines known to software to be modified and only in one cache can make use of an optimization under $SW_{cc} \Rightarrow HW_{cc}$ transition when no false sharing of the line exists. We can add an operation to allocate an entry in the directory with the L2 holding the line set as owner. The operation removes the need for invalidation and acknowledgment messages to be sent. Such an optimization adds complexity to the protocol and therefore was not evaluated in this work. The optimization requires holding a modified line in the L2 while the directory update occurs, which may be an unwanted design constraint. Without the constraints, a cache eviction of a modified line could race with the directory update, resulting in the directory holding an inconsistent value, i.e., a dirty line held in a cache when in fact no cache holds the line. There is precedence for such an operation. The use of a specialized output write operation is similar to write-back commit optimization used in some hardware transactional memory implementations, such as [108], where a commit in our case is a $SW_{cc} \Rightarrow HW_{cc}$ transition. The optimization makes data visible, i.e., places it back into the coherence protocol, without forcing a cache eviction or transmitting data.

Similarly, widely read-shared data that is HW_{cc} is made SW_{cc} trivially if the

sharer count drops to zero, which can occur when all shares invalidate the line in the L2 or the clean line is evicted from the L2. Under SW_{cc}, this should be the case at a synchronization point. Similar optimizations could be applied to protocols that have exclusive or owned states allowing a single owner in HW_{cc} to acquire sole ownership in SW_{cc}. When the transition is requested, if the sharer count is zero, the directory incurs a miss and thus requires no coherence probes to be sent. The state change request is acknowledged immediately.

8.4 Software Use Cases

In this section, we discuss the use of objects that use COHESION to interleave coherent and non-coherent data as a natural extension of object-based synchronization. We discuss the problem facing contemporary accelerators that try to adopt applications and programming models that assume hardware coherence. We discuss how contemporary programming patterns can be supported by COHESION. We conclude by discussing two issues not addressed in this dissertation that have potential as future work: (1) quantifying the design complexity of a system with both HW_{cc} and SW_{cc} and (2) the difficulty in determining the appropriate amount of hardware to dedicate toward supporting HW_{cc} on a hybrid memory model architecture.

A general pattern where we see COHESION being useful is for mixed-coherence-domain objects. These are objects where some data must be kept coherent, but other data is amenable to software-managed coherence. Using the fine-grained region table, it is possible to allocate objects that leave some fields hardware coherent while others remain untracked by hardware. A simple example where this would be used is for keeping a reference count on a read-only data structure. The count must be kept coherent, but the data is immutable. In COHESION, the

object would be allocated as incoherent and the counter value would be placed alone on a line and marked as coherent.

A problem exists with accelerator systems today. Applications with data structures or compute patterns that heavily leverage hardware cache coherence are unable to easily exploit the parallel processing power of available accelerators, e.g. GPUs, if at all. For those applications that are amenable to GPU acceleration, but were developed with programming models where hardware cache coherence was assumed, a rewrite is required. For software that has a data-parallel kernel without need for coherence, but requires a host application that may find coherence beneficial, a similar problem exists. For applications with shared irregular data structures that require irregular updates, such as graphs or kd-trees, there may be no way to efficiently support their execution without hardware cache coherence. To make accelerators and future general-purpose systems with large numbers of cores efficient platforms for applications, there must be a means to support hardware cache coherence at some level.

Having the ability to support current programming models that require hardware coherence will open up future platforms to more developers. The key is to make coherence management an optimization choice instead of making it a burden for correctness. For applications that use task stealing, mutexes, reductions, mailboxes, and shared data regions, COHESION has the ability to support those constructs efficiently with its HW_{cc} protocol while using the SW_{cc} protocol for the data-parallel regions of execution.

While the industry standard OpenCL [30] model does not require coherence, it offers the opportunity to greatly simplify host-device semantics and lower performance overhead for heterogeneous systems, while making software targeting a compatible platform portable. By providing some amount of coherence, shared data can be made resident in local memories, which would be caches in this case,

without the need for explicit programmer intervention. Performance could be increased by removing the explicit copy operation from host to device memory; on a heterogeneous system with CPU cores and accelerator cores with a single address space supporting a hybrid memory model, there is no such distinction.

There are implementation concerns for COHESION not fully addressed in this dissertation. One is the additional cost of designing two protocols for one processor, i.e., a HW_{cc} protocol implementation and a SW_{cc} protocol implementation. While it is true that designing a hardware coherence implementation is difficult, we argue that COHESION reduces the importance of having a highly optimized coherence protocol since the goal in developing highly parallel software for a hybrid memory model architecture is to reduce the use of coherence to its bare minimum. Thus, the performance of the system will be less sensitive to hardware coherence overhead than a comparable system that only supports hardware coherence. The marginal hardware design cost of adding support for software-managed coherence is minimal. There are extra bits that need to be kept at the L2 and L3 caches in our design and a small amount of merging logic kept at the L3. Otherwise, most of the complexity of implementing a software-managed coherence protocol is embedded in the runtime or the compiler and not in the architecture.

The one additional concern for an architecture supporting COHESION is how to choose an allocation of hardware coherence resources. In a conventional HW_{cc}-only design, the size of the directories or capacity of the network is determined by the number of lines that must be tracked by the coherence protocol and the amount of coherence traffic expected. For COHESION, those characteristics can vary between applications and depend greatly on how much optimization has been performed by software. For some workloads, there may be almost no use of HW_{cc}, while others depend on it heavily. The promise of COHESION is to make scalable applications perform correctly with little effort and achieve high performance when

optimization efforts are focused on removing the use of HW_{cc}, all while using modest hardware. Therefore, we believe that the area overhead thresholds of approximately 10% that we use in evaluating this work are reasonable, based on our performance evaluations. However, there is no clear guiding principle for choosing this threshold, and more hybrid memory model systems may need to be built before any generalizations can be made.

8.5 Programming Examples

In this section we present high-level programming examples that use COHESION. We break the examples into three categories, which represent the predominate patterns used by COHESION applications. Static COHESION divides the address space of the application into HW_{cc} and SW_{cc} partitions at initialization. Throughout the runtime of the application, this partition does not change. Dynamic COHESION supports the migration of fine-grained regions of memory between coherence domains throughout the runtime of the application. System software is distinct from the static and dynamic patterns since it is not meant for applications software, but is a pattern that is found in runtimes, synchronization libraries, and operating systems. We now go through each pattern in detail.

8.5.1 Static Partitioning

For many of our workloads, the data can be partitioned into that which would best be served by HW_{cc} and that which would best be served by SW_{cc}. Static partitioning is similar to systems with multiple address spaces where one is coherent. However, since COHESION covers the entire address space, COHESION adds the advantage of allowing software to choose the partition. A trivial case for static partitioning would put all code and stack data into SW_{cc} and put the rest

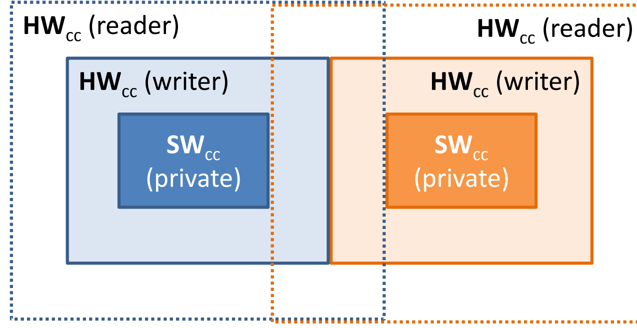


Figure 8.5: The static COHESION pattern demonstrated for a 2D stencil computation. The goal is to place the per-cell private regions in the SW_{cc} domain and only use HW_{cc} domain for a small border region where communication is necessary.

into HW_{cc}. More elaborate partitioning could be performed based on compiler analysis for read-write sets and liveness.

Figure 8.5 demonstrates an example of static partitioning. The figure shows two grid cells from a 2D stencil computation such as our `heat` benchmark. Each task is allocated a cell and uses two buffers, one to read from, and one to write to, in each of a number of time steps. A vast majority of the data is only accessed by a single task and therefore need not be kept coherent. The boundary regions are shared across tasks. To enable such sharing trivially, we can statically partition the dataset of each task to have the private data kept SW_{cc} and the shared perimeter data kept HW_{cc}. The advantages of this approach are that only the truly shared data must pay the cost of coherence and that the software must only change trivially to support COHESION.

8.5.2 Dynamic Partitioning

An advantage of COHESION over systems that support only static partitioning is the ability to transition between coherence domains at runtime. COHESION achieves performance advantages when transitions are optimally placed by minimizing the amount of data that must be tracked by the coherence protocol at

any instant in time. Moreover, COHESION is able to use the entire address space and can be virtualized. If only static partitioning is allowed, the developer is left with a fixed partition that he must manage explicitly. Moreover, having only static partitions would require splitting and reorganizing data structures across HW_{cc} and SW_{cc} domains. The need to reorganize data structures can cause large amounts of code to be re-factored, which is undesirable. COHESION avoids these programmability and performance bottlenecks by using fine-grained remapping between coherence domains that can occur at runtime.

Figure 8.6 provides an example of dynamic partitioning. The figure shows four cores performing a divide-and-conquer sort operation. During the divide phase, COHESION is used to distribute a partition of the working set to another processor using HW_{cc} while the processor performing the pivot keeps the data in SW_{cc}. Once the processors reach the sequential phase of the sort, they can rely on SW_{cc} alone to avoid any coherence costs since there is no sharing. When the results are available, the processors can simply place them in the HW_{cc} domain, which allows consumers to source the data and avoids cache flushing.

Another example includes ray tracing where a spatial data structure, such as a kd-tree, is built during one phase and consumed in a mostly data-parallel fashion during another phase. More advanced implementation may allow updates while reading. These cases can be handled by forcing the data HW_{cc} before an update and then transitioning back to SW_{cc} after the update.

8.5.3 System Software

System software is aided by COHESION. Software-managed coherence does not allow for migration of tasks that may have modified data in the local caches. Moreover, after a task or application completes, there is no way to assure all data

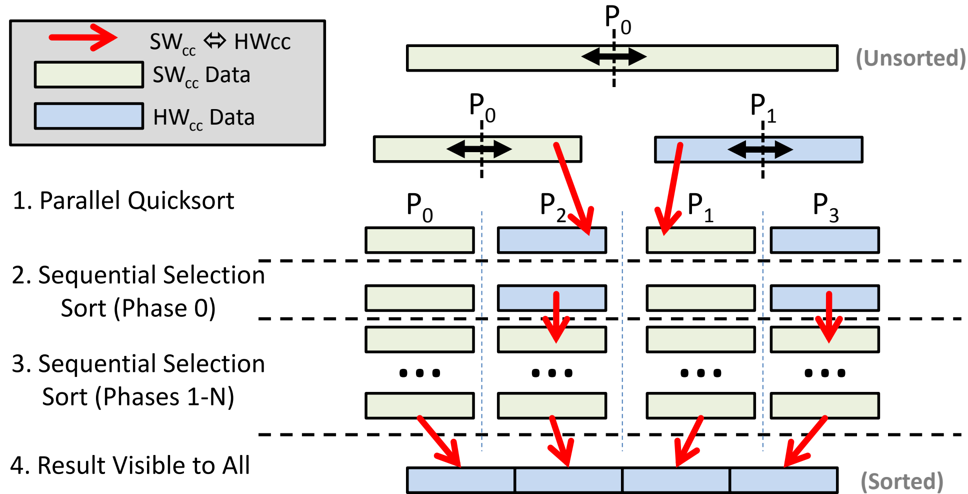


Figure 8.6: The dynamic COHESION pattern demonstrated using a two-phase parallel sort on four processor cores. The goal is to use HW_{cc} as-needed during the divide phase and avoid HW_{cc} during periods of independent execution, shown as the serial sort in the figure. When the results are ready, HW_{cc} is used to make the produced data available to consumers.

touched by the tasks is consistent with the L3 and memory without flushing the entire shared cache. Having the ability to force data into a hardware-tracked state, albeit at some cost, is an advantage of COHESION. Moreover, should the runtime need to preempt a task and migrate it, the task's working set can be made HW_{cc} and the data will migrate with the task.

As described in Chapter 3, the runtime on the baseline Rigel system required that all shared data be flushed from local caches before being made globally visible. The implication was that the runtime had to be conservative and would force accesses to go to the L3 to read data that may have been produced locally. Moreover, it was unclear how to design a runtime that would support work stealing or task migration without the use of coherence. For short-lived tasks, migration was less of an issue. However, work stealing may have proved a better option as was shown in a study done using a similar runtime implemented on a system with coherence support [78].

8.6 Compatibility

Supporting COHESION raises the concern of forward compatibility. A valuable and often necessary requirement for architectures is that software written for the current platform can be run on future derivative platforms. While it is not always critical that software run faster, it is desirable that the software not run significantly slower on future platforms. More importantly, to achieve compatibility software must not need source- or binary-level modification to be correct across generations. We now discuss some of the design requirements so COHESION can be made to run correctly across different platform configurations.

For systems supporting COHESION, there is the question of how software-managed coherence will work if the size and number of levels of the cache are changed across generations. The key design attribute of COHESION that allows future platforms to continue supporting COHESION-enabled software from previous versions of the platform is the lack of *guaranteed incoherence*. The software-managed coherence scheme used in COHESION, the Task-Centric Memory Model, does not allow for two caches to hold different values for the same address. Defined inconsistencies across local memories is permissible by design in systems with software-managed local memories, such as GPUs, where software alone manages the contents of the local memories. However, our model is based on hardware caches and supports a shared single address space thus avoiding incoherence by design. While it is possible to write software in defiance of the Task-Centric Memory Model on the baseline architecture, it is disallowed by our software implementation. As such, if the size of caches grows or shrinks across generations, there will be no problem of incoherence being lost since it is disallowed in current implementations.

Another concern is running COHESION-enabled software on architectures that

only support hardware-managed coherence such as contemporary CMPs. In such cases, the appearance of a coherent single address space is maintained trivially in hardware. The software management operations are used as hints or can simply be disregarded by the hardware, and hardware-managed coherence is used instead. Furthermore, the writeback and invalidate operations that we use in our baseline architecture are supported by analogous cache-management operations in most contemporary instruction set architectures. Thus even ISA-level backwards compatibility could be added to contemporary ISAs extended for use with Cohesion.

Adding extra levels of cache hierarchy can represent trade-offs in the implementation of a hybrid memory model. As new levels are added, software may be required to manage data movement to achieve better performance if the hierarchy is coherent. Moreover, application software using a software-managed coherence scheme may need to reconsider correctness if new levels are added to the cache hierarchy. For this work, we assume that Cohesion must operate between two levels of cache: the cluster caches (L2) and the global caches (L3). The global cache is the point of coherence for the processor. If the value in the L3 is consistent with all values in the L2, any read on the chip will obtain a consistent copy. If an extra level of cache is inserted between the L2 and L3, coherence for the software-managed coherence scheme requires that writebacks and invalidates force values not only from the L2, as is true in our implementation, but also from the intermediate level between the L2 and L3. More complex software-managed coherence protocols that take into account more than two levels of cache that must be managed by the protocol are possible, but not evaluated in this work. The Task-Centric Memory Model will continue to provide coherence regardless of the number of levels of cache as long as the coherence management operations invalidate or writeback lines up to the level of cache serving as the point of

coherence.

8.7 Summary

In this section we presented Cohesion, a scheme for enabling coherence domain transitions at runtime. Cohesion is a proof-of-principle implementation of a hybrid memory model. We described the baseline software and hardware coherence protocols used in this work. We then described the protocols necessary for transitioning between the software-managed coherence domain and the hardware-managed coherence domain. We discussed the basic software interface to Cohesion and a simple implementation of its hardware mechanisms. We concluded by discussing potential optimizations to the Cohesion design presented in this section. The end result is a 1000-core chip that exports a conventional memory model to developers while incorporating heterogeneous coherence mechanisms. It provides accelerator-like scalability while supporting a conventional general-purpose memory model.

CHAPTER 9

Evaluation

In this section we evaluate the memory model and related protocols and mechanisms presented in this work. We first discuss our methodology, including our benchmark suite and simulation infrastructure. We evaluate the Task-Centric Memory Model by executing various policies that determine when coherence actions occur, and we evaluate the efficiency of the software coherence actions. WayPoint, our mechanism for lightweight hardware coherence, is evaluated. WayPoint is shown to reduce the on-die directory overhead and associativity needs with little performance impact relative to an aggressive and high-overhead design. Lastly, we evaluate the initial implementation of a hybrid memory model, Cohesion.

9.1 Methodology

We simulate the architecture described in Chapter 3 using RigelSim, our execution-driven 1024-core simulator of the Rigel processor. The parameters used for runs without hardware cache coherence are listed in Table 9.1. The additional parameters relevant to simulations with cache coherence enabled are listed in Table 9.2. We use an execution-driven simulation of the design and run each benchmark for at least one billion instructions after initialization. We model cores, caches, interconnects, and memory controllers. The cycle-accurate DRAM model and memory controller use the GDDR5 timings listed in Table 9.3. Our simulator is

Table 9.1: Timing parameters for the baseline architecture.

Parameter	Value	Unit	Parameter	Value	Unit
Cores	1024	–	Line Size	32	bytes
Memory BW	192	GB/s	Core Freq.	1.5	GHz
DRAM Channels	8	–	DRAM Type	GDDR5	–
L1I Size	2	kB	L1I Assoc.	2	way
L1D Size	1	kB	L1D Assoc.	2	way
L2 Size	64	kB	L2 Assoc.	16	way
L2 Size (Total)	8	MB	L2 Latency	4	clks
L3 Size	4	MB	L3 Assoc.	8	way
L2 Ports	2	R/W	L3 Ports	1	R/W
L3 Latency	16+	clks	L3 Banks	32	–

Table 9.2: Additional sizing and timing parameters for Rigel with cache coherence.

COHESION			Optimistic Directory		
Parameter	**Value**	**Unit**	**Parameter**	**Value**	**Unit**
Directory Access Lat.	2	Cycles	Directory Access Lat.	2	Cycles
Directory Size	16K	$\frac{entries}{L3\ bank}$	Directory Size	∞	$\frac{entries}{L3\ bank}$
Directory Assoc.	128	ways	Directory Assoc.	Full	–

structural in that there are analogues in hardware for each software component used for timing. Interfaces to these blocks are meant to approximate an RTL implementation.

The applications that we evaluate are optimized kernels extracted from scientific and visual computing applications. The SW_{cc} variants have explicit invalidate and writeback instructions in the code at task boundaries. The COHESION variants have such instructions when SW_{cc} is used and none for data in the HW_{cc} domain. The COHESION API is used to allocate the SW_{cc} and non-coherent data on the incoherent heap. The HW_{cc} versions eliminate programmed coherence actions in the benchmark code. The benchmarks are written using a task-based, barrier-synchronized work queue model. The kernels include conjugate gradient linear solver (`cg`), dense matrix multiply (`dmm`), 2D fast Fourier transform (`fft`), collision detection (`gjk`), 2D stencil (`heat`), K-means clustering (`kmeans`), march-

Table 9.3: GDDR5 DRAM memory timings used in our simulations. All units are DRAM cycles assuming 3 GHz DDR at 6.0 Gbps per pin.

Parameter	**Timing**	**Parameter**	**Timing**
t_{RAS}	44-48	t_{RC}	61-66
t_{RRD}	5-7	t_{FAW}	22-25
t_{RCD}R	20-24	t_{RCD}W	11-14
t_{RP}	16-19	t_{CL}	16-20
t_{WR}	20-23	t_{WTR}	8-12
t_{WL}	2	t_{RTP}	3-5
t_{CCD}L	3-4		

ing cubes (`march`), medical image reconstruction (`mri`), edge detection (`sobel`), and 3D stencil (`stencil`). Complete descriptions of the benchmarks are available in Chapter 4.

9.2 Results: Task-Centric Memory Model

In this section, we evaluate the Task-Centric Memory Model using an implementation of the Rigel Task Model running on RigelSim. In this section, we demonstrate two key results. The first is that the overhead of software-enforced coherence is less than 10% in most cases compared to an optimistic hardware-coherent baseline. Eager software-managed coherence actions, which are performed at task completion, can even improve performance in other cases by reducing instantaneous bandwidth demands placed on the system at barriers. The second result is that a common hardware optimization, hardware prefetching, is highly beneficial to performance when performed by the global cache, a case allowed trivially by our model; but it is of questionable benefit when performed at the cluster cache, the case that is handled with difficulty by software-managed cache coherence alone.

Table 9.4: Overview of coherence management policies for TCMM.

Mode	Description	Patterns
EIEW	Invalidate and writeback at task completion	Low reuse/sharing of read-input data, need to overlap output writes with execution
EILW	Invalidate at task completion, writeback before barrier	Low reuse/sharing read-input data, reuse of written data
LIEW	Invalidate before barrier, writeback at end of task	High read sharing and read reuse, overlaps output writes with execution
LILW	Invalidate and writeback before barrier	Large reuse and sharing sets for all data, may simply flush caches at barriers

9.2.1 Policy Selection

Our implementation permits coherence actions to be performed lazily at barriers or eagerly at task boundaries. Moreover, we can split writeback operations that write a dirty line back to the last-level cache from invalidate operations that set a clean line in the private caches to invalid. In Figure 9.1, we evaluate eager invalidate-eager writeback (EIEW), lazy invalidate-eager writeback (LIEW), eager invalidate-lazy writeback (EILW), and lazy invalidate-lazy writeback (LILW) relative to the optimistic baseline that removes all writebacks and invalidates and provides correct values without coherence traffic. For reference, Table 9.4 shows the different policies and how they are used. The results show that different policies provide the best performance for each benchmark. Only one benchmark (`mri`) suffers greater than 10% overhead relative to an optimistic zero-cost hardware coherence baseline due to software coherence actions.

The reason for the high overhead for `mri` is likely due to the large volume of data touched combined with long task lengths. Even for eager policies, there are a large number of writeback and invalidates that much be executed for the

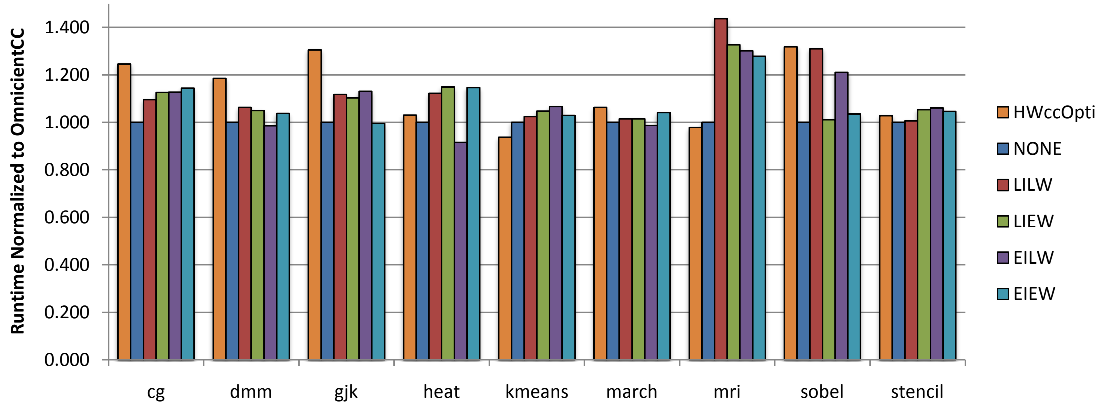


Figure 9.1: Runtime for different eviction policies compared to a highly optimistic hardware coherence scheme. Software schemes can be selected on a per-application basis. OmniscientCC corresponds to an implementation with hardware coherence disabled, no coherence traffic for writebacks or invalidates, and an omniscient memory model that provides correct values.

benchmark, many of which have low utility, as we show in a later section. One other benchmark-specific note is `heat`. For this benchmark, eager invalidates improve performance noticeably. The reason for the effect is that `heat` has large input and output sets that are touch-once data inter-task. The eager invalidation policies have the advantage of providing a better replacement policy at the L2 cache by evicting streaming data in lieu of older data that may be used again.

Since the model is under software control, a mix of policies across applications can be deployed. In general we find two trends. First, eager writebacks overlap write traffic with useful execution and should be used as much as possible to increase memory system concurrency. The coherence actions result in less bursty load on the interconnect, increasing performance. Brewer and Kuszmaul observe a similar effect due to output port contention on the CM-5 [109]. Second, lazy invalidation allows for shared read-input data to be exploited opportunistically when two tasks share read values and execute on the same core, or in the same cluster on Rigel, during an interval.

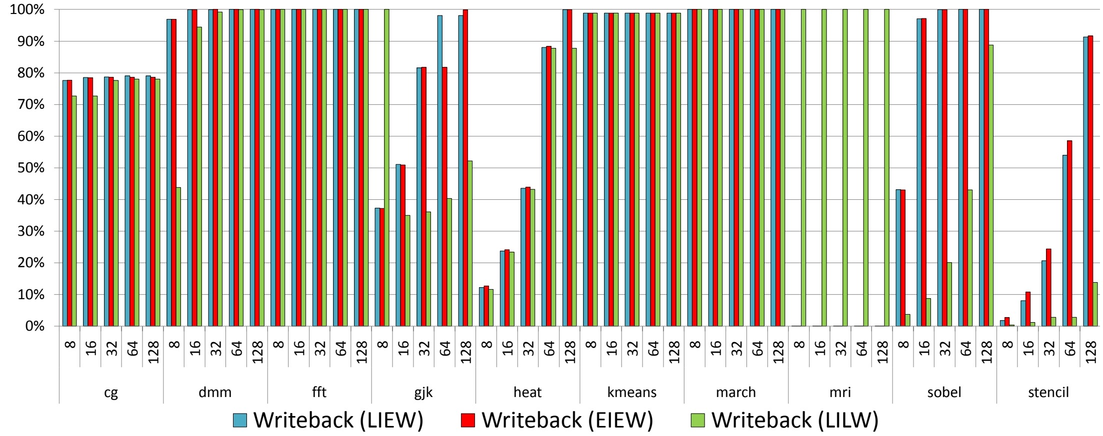


Figure 9.2: The impact of L2 (cluster) cache sizing on writeback efficiency.

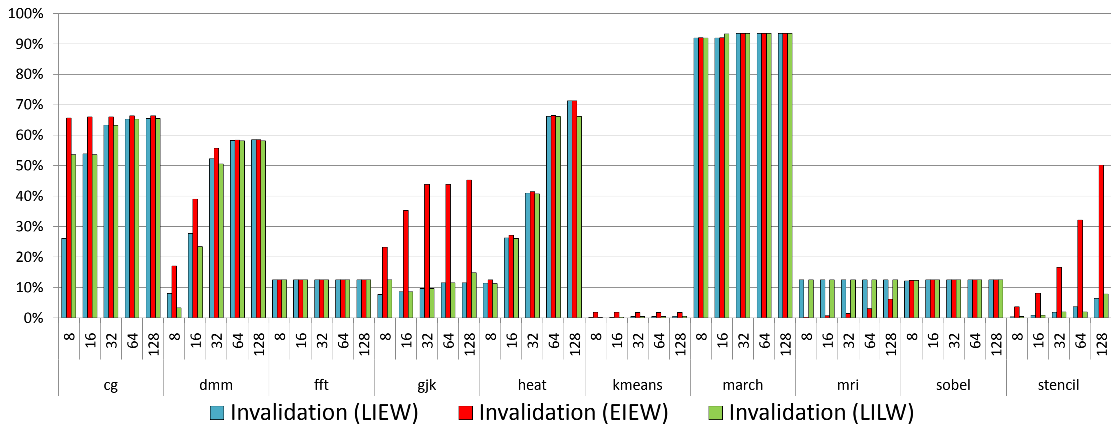


Figure 9.3: The impact of L2 (cluster) cache sizing on invalidate efficiency.

9.2.2 Software Coherence Action Utility

Figure 9.2 and Figure 9.3 illustrate the utility of writeback and invalidation operations for the Task-Centric Memory Model, respectively. We define utility as the number of invalidation or writeback instructions that are issued to lines valid in the cache at the time of the issue. Writebacks with utility force an action to occur at the L2 while those without utility are nops. A larger utility implies that fewer wasted instructions are issued and that TCMM is efficient. For cases where the writebacks and invalidations have low utility, TCMM is forced to be overly conservative and is wasting fetch and issue bandwidth on useless instructions.

The results demonstrate four general trends. First, writeback efficiency is high for most workloads and policies. Second, invalidate utility is much lower. This effect is likely due to most applications performing writes to output data near the end of the task while reads are performed closer to the start. The reduction in time between last access and coherence action increases the probability that an action has utility. Third, eager policies, which invalidate or writeback data at the end of a task instead of waiting until the end of an interval, have higher utility. Moreover, eager policies require less bookkeeping since they only require tracking coherence state over the length of a task and not the entire interval. Finally, efficiency tends to increase with cache size, which is due to the fact that conflicts and capacity misses that generate evictions are less likely to occur between the last access to a line and the coherence action.

For roughly half the benchmarks, utility is below 15%. The low level of utility for explicit coherence actions indicates that instruction issue bandwidth and tracking state are being wasted on lines that are evicted from the cache due to implicit causes, e.g., cache eviction due to a capacity miss. The extra actions cannot be eliminated a priori in software in a trivial, way and these actions must be present to ensure correctness.

The conservatism required results in an overhead for SW_{cc} that we would like to avoid. While not evaluated in this work, there are opportunities for hardware structures at the L2 cache and additional bits associated with each L2 line to reduce the need to issue unnecessary cache management operations. Better software techniques, either compiler-driven or developer-driven, could potentially lower the overhead. Finally, a trivial solution is to simply flush the caches at a barrier if the number of writebacks and invalidates grows to be too high.

We find that even with the increased overhead, software-managed coherence can be a low-overhead coherence scheme for 1000-core processors. However, we

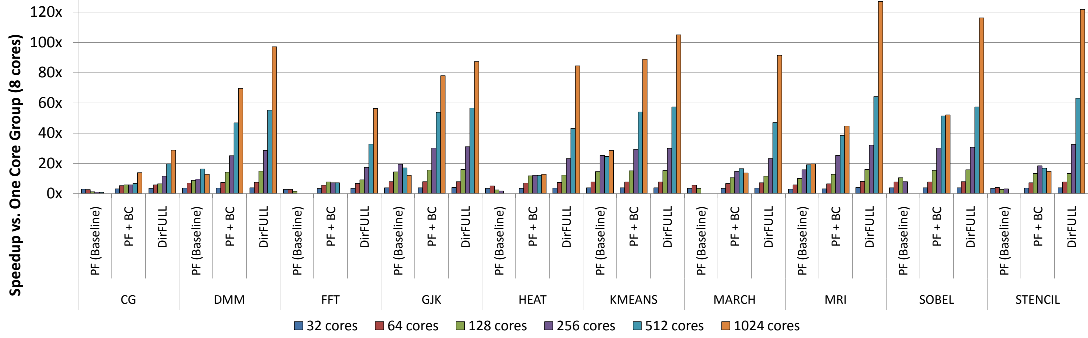


Figure 9.4: Scalability of baseline probe filtering with 2048 entries compared to SPF with broadcast-collective support and an on-die full directory. Perfect speedup would be 128× in this figure.

do see an opportunity for adding mechanisms in hardware and tools support that could reduce the inefficiencies in software-managed coherence. In this work, we explore one mechanism, COHESION, which allows for data to be moved into the hardware coherence domain when software management is inappropriate, thus reducing one of the causes of inefficiencies in software-managed coherence.

9.3 Results: Scalable Probe Filtering

Here we evaluate the performance of the probe filter with and without a broadcast network for accelerating coherence messages. We use 2048-entry fully associative probe filter cache (PFC) banks to minimize the effects of conflict and capacity misses. The result, as shown in Figure 9.4, is that a PFC alone is not generally scalable, but with SPF scalability can be extended. We find that rebuilding sharing vectors in the collective network can improve scalability over an invalidation-based policy, but that probe filter misses still limit scalability.

One potential solution is to build larger PFCs. However, building larger PFCs is confounded by area and power constraints and the large datasets of throughput-oriented and streaming workloads. Coarse-grained tracking mechanisms such as

RegionScout [63] decrease PFC area overhead, but may have limited utility for workloads with low spatial locality.

The essential problem with the probe filtering approach is that sharer state that is not explicitly tracked, such as in a directory entry, must be handled conservatively on the first access, necessitating broadcasts. The power and performance costs of broadcasts can be reduced with techniques such as SPF that reduce the number of messages with additional complexity, area, and power overhead, but scalability is still limited. To eliminate PFC miss broadcasts all together, we next evaluate WayPoint, which captures the performance benefits of exact sharing information with small hardware structures.

9.4 Results: WayPoint

Here we evaluate the performance of WayPoint. We use a subset directory cache implementation with varying sizes and associativities for comparison. We also use an optimistic, albeit unrealizable, full on-die directory to serve as a baseline. The optimistic full directory is infinitely sized, thus removing all but cold-cache misses from the performance overhead of the baseline.

9.4.1 Directory Cache Sizing

Figure 9.5 shows the runtime of the benchmarks normalized to the optimistic coherence baseline. The data on the left are configurations without WayPoint where invalidations occur when the directory cache capacity is reached. The data on the right are for configurations with WayPoint. An eviction with WayPoint results in an entry being placed in an overflow list in cached memory, thus avoiding the overhead of invalidations. Note that the directory sizes, measured in entries per L3 bank, are not equal between the two configurations because the non-

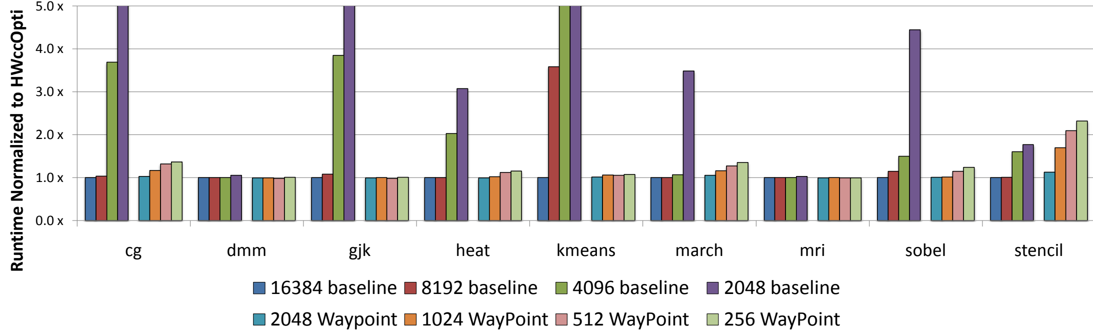


Figure 9.5: The runtime normalized to the optimistic hardware cache coherence implementation is shown for different sizes of directory cache. Configurations without WayPoint are shown on the left and those with it on the right.

WayPoint configurations slow down precipitously with fewer than 2048 directory entries per bank.

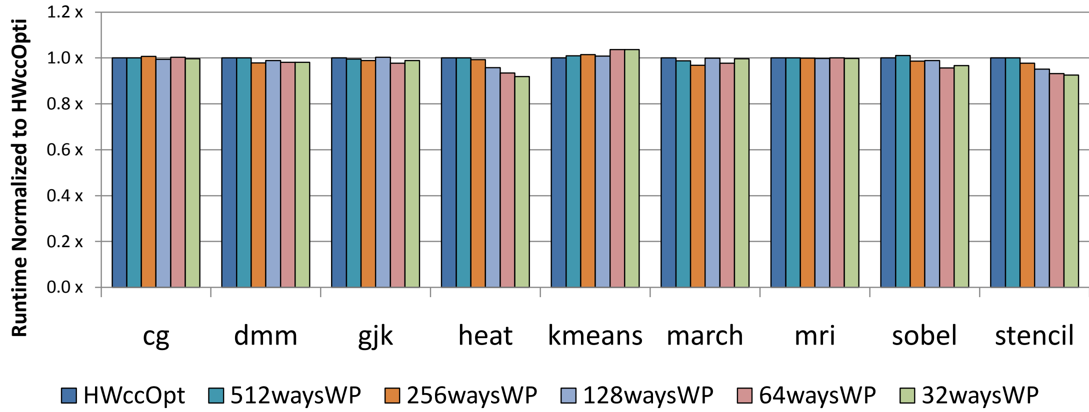
The results in Figure 9.5 indicate that at least 8192 entries per bank would be needed to eliminate large slowdowns when using a conventional directory cache. With WayPoint, that number can be 2048 or fewer, thus providing comparable performance across all benchmarks with over 4× fewer resources. Moreover, performance degradation for designs with WayPoint follows a much smoother curve leading us to believe that even for pathological workloads that we may not have evaluated, WayPoint should be far more robust to workload variation than configurations without WayPoint. Note that the one benchmark with appreciable overhead for WayPoint is `stencil`. The accesses for this benchmark tend to have little locality and little reuse since the benchmark streams through every output once and each input a small number of times each iteration. Furthermore, the read sharing is limited. Therefore, `stencil` tends to thrash in the directory cache more than the other workloads. Even so, the overhead for `stencil` is much less with a directory cache supplemented by WayPoint than without WayPoint.

9.4.2 Directory Cache Associativity

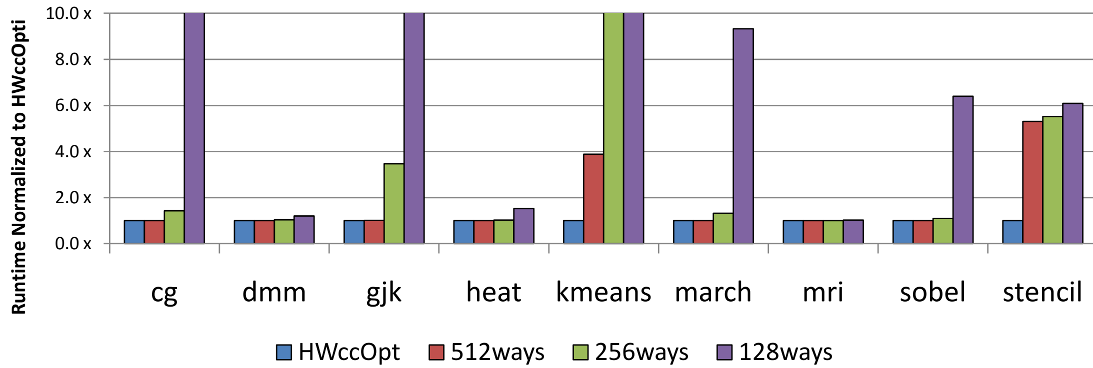
Figures 9.6(a) and 9.6(b) give the execution time with and without WayPoint while varying the associativity of a fixed-sized directory. The goal of these experiments is to demonstrate the ability of WayPoint to reduce the associativity demands of the directory cache. All results vary the number of ways and sets inversely to maintain a constant 16384-entry directory cache except for the infinite-sized on-die directory baseline, which has one set and unlimited ways. We show 32–512 way directory caches with WayPoint and 128–512 way without WayPoint. The discrepancy is due to the greatly increased runtime due to thrashing and subsequent invalidations for lower associativity directory caches without WayPoint.

The results show that for highly associative directory caches, the configurations perform comparably. However, for the configurations without WayPoint, there is a clear performance cliff where runtime increases dramatically below a threshold associativity. The performance versus associativity curve is smoother for WayPoint. With WayPoint, performance varies by less than 10% across all configurations. We find that some benchmarks have *critical associativities* where performance is constant while reducing associativity until a threshold is reached. The benchmarks `cg`, `gjk`, `march`, `sobel`, and `stencil` exhibit critical associativities. The reason for this performance cliff is that these benchmarks have working sets that are spread evenly across caches and have a regular access pattern, which means many lines in lower-level caches map to the same bank of the last-level cache and its slice of the directory cache. When the number of lines per cache times the number of caches exceeds the available directory associativity, thrashing occurs, thus greatly reducing performance.

Also note that the results we present in this section have had optimizations



(a) With WayPoint



(b) Without WayPoint

Figure 9.6: Runtime of WayPoint-enabled simulations with different associativities with fixed directory size. Results normalized to optimistic hardware cache coherence. Note that we have a one-to-one correspondence between sets and WayPoint lists in (a), resulting in slightly less contention and thus better performance for less-associative on-die caches.

performed to the software to remove conflicts at the directory. A deeper discussion of the software optimizations is given in Chapter 7. What is important to note is that without these optimizations, the results for configurations without WayPoint would have much higher runtimes. Some of the configurations had such a high degree of thrashing that simulation became intractable.

9.4.3 Power and Area Estimates

We evaluate the area required for our directory cache architecture with and without WayPoint and show the results in Table 9.5. Each of the 32 L3 banks is assigned a disjoint region of the address space, and has a corresponding directory cache bank that handles the same address space region. We use CACTI 6.5 [110] to estimate area and power of the directory caches. We evaluate a high-performance 45 nm process at 1.2 GHz with two-cycle pipelined accesses and separate read and write ports. We assume a total die area of 300 mm^2 based on the analysis performed in [42]. We use average activity factors from our timing simulator to estimate dynamic power usage.

The results show that the power and area efficiency of the structures degrades linearly with the degree of associativity; these results motivate a mechanism such as WayPoint, which provides good performance with a low-associativity directory cache. Moreover, we find that directory cache capacity can be doubled at a much lower marginal power and area cost than can associativity—generally less than a 2× area and power increase for every 2× capacity increase. However, we have found that the marginal utility of more than 2048 entries per bank to be small with WayPoint. Note that while Table 9.5 shows the area overhead of a 64-way directory cache to be sizeable (23.4% of chip die area), Figure 9.6(b) indicates that even at a much higher capacity, four times that associativity would

Table 9.5: Power and area estimates for a 2048-entry WayPoint implementation.

Configuration		Bank Dimension (mm)		Combined Area		Combined Power (W)		
Sets	Ways	Width	Height	mm^2	Chip Area	Static	Dynamic	Total
512	4	0.567	0.484	8.78	2.93%	1.48	0.15	1.63
256	8	0.396	0.878	11.14	3.71%	1.64	0.23	1.87
128	16	0.324	1.660	17.23	5.74%	1.99	0.43	2.42
64	32	0.305	3.323	32.43	10.8%	2.74	0.75	3.50
32	64	0.341	6.596	71.93	23.4%	4.94	1.44	6.38

be required to approach the performance of a WayPoint, which requires greatly reduced hardware overheads.

9.5 Results: Cohesion

We evaluate four design points: SW_{cc}, optimistic HW_{cc}, HW_{cc} with realistic hardware assumptions, and Cohesion with the same realistic hardware assumptions. The hardware configuration for SW_{cc} is equivalent to our baseline described in Chapter 5, but with no directory and all sharing handled by software. For SW_{cc}, writes occur at the L2 with no delay, and evictions performed to clean data happen without creating any network traffic. The optimistic HW_{cc} case removes all directory conflicts by making the on-die directory infinitely sized and fully-associative. The behavior is equivalent to a full-map directory in memory [67], but with single-cycle latency to the directory. The HW_{cc} realistic case comprises a 16k entry, 128-way sparse directory at each L3 cache bank. The Cohesion configuration uses the same hardware as the realistic HW_{cc} configurations.

9.5.1 Message Reduction

Figure 9.7 gives the number of messages sent by the L2s to the directory, normalized to SW_{cc}. There is a reduction in messages relative to the HW_{cc} configurations across all benchmarks. `kmeans` is the only benchmark where SW_{cc} shows higher

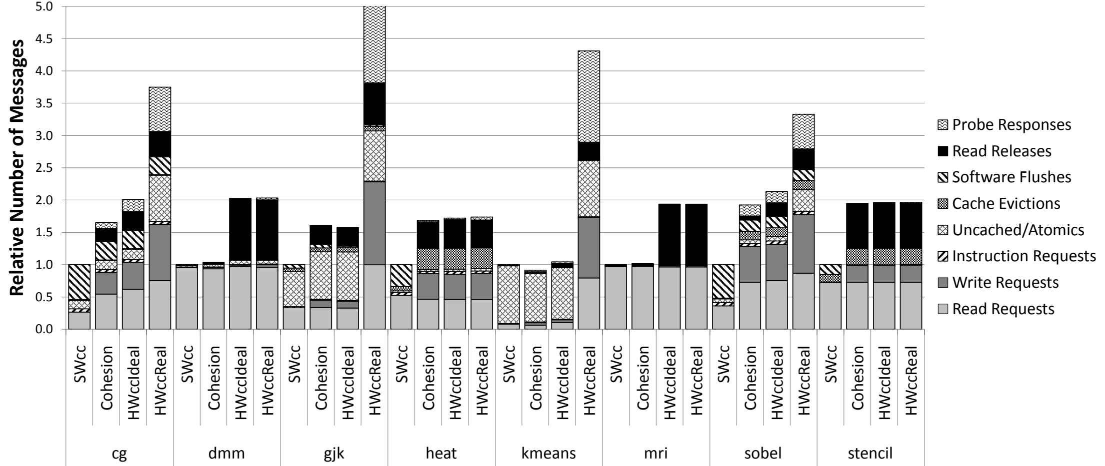


Figure 9.7: Number of messages sent out of the L2 (cluster) cache.

message counts than COHESION. This reduction is due to optimizations that reduce the number of uncached operations issued by the benchmark by relying upon HW_{cc} under COHESION. For some benchmarks, the number of messages is nearly identical across COHESION and optimistic HW_{cc} configurations, such as `heat` and `stencil`. We see potential to remove many of these messages by applying further, albeit more complicated, optimization strategies using COHESION. While we leave more elaborate coherence domain remapping strategies to future work, our initial efforts here show that combining SW_{cc} and HW_{cc} can greatly reduce the number of network messages. The benefits include less contention, resulting in lower latency for network messages and the opportunity for architects to reduce network costs while maintaining the same level of performance.

9.5.2 Directory Entry Savings

Figures 9.8 (a) and 9.8 (b) show the normalized runtime for different directory sizes under HW_{cc} and COHESION, respectively, compared to an infinite-sized directory. We make directories fully associative to isolate the influence of capacity. The rapid drop-off in performance shown in Figure 9.8 (a) demonstrates the sensitivity of

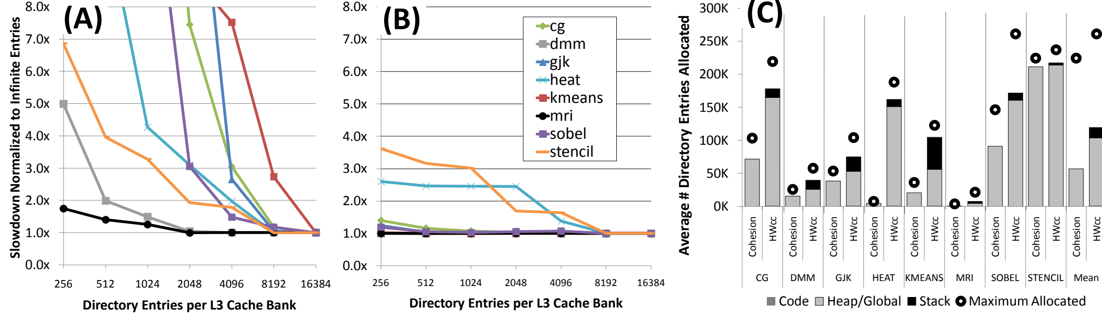


Figure 9.8: Runtime with different directory cache sizes (in thousands) for our baseline without COHESION (a) and with COHESION (b). Part (c) shows the average and maximum number of directory entries used at runtime out of the total number of entries.

$HW_{cc}$to directory cache sizing. The explosion in runtime stems from capacity misses generating evictions at the directory cache, thus triggering L2 invalidates. Figure 9.8 (b) shows that COHESION reduces the performance sensitivity with respect to directory sizing across all benchmarks. Figure 9.8 (b) indicates a greatly reduced sensitivity to smaller directory cache sizes, allowing for directory cache resources to be reduced without impacting performance.

Figure 9.8 (c) shows the mean and maximum number of directory entries used by COHESION and the optimistic HW_{cc} baseline. We average samples taken every 1000 cycles. We classify the entries as to whether they map to code, which is negligible, private stack data, or heap allocations and static global data. The HW_{cc} data in Figure 9.8 (c) is a proxy for the on-die working set of the benchmarks, because all lines cached in an L2 have allocated directory entries and all uncached lines are removed from the directory when the L2 sharer count drops to zero. The data also demonstrates the degree of read sharing, because the ratio of directory entries to valid L2 cache lines is smaller when there are more sharers and thus fewer directory entries are needed.

Across all benchmarks, COHESION provides a reduction in average directory utilization of 2.1×. For some benchmarks, simply keeping the stack incoherent

achieves most of the benefit, but, on average, the stack alone only represents 15% of the directory resources. Code makes up a trivial portion of the directory entries because our benchmarks have large datasets, but code entries may be evicted by touch-once data when streaming through large datasets, thus impacting performance. These results show that most of the savings comes from using COHESION to allocate globally shared data on the incoherent heap, thus avoiding coherence overhead for that data.

9.5.3 Directory Area Estimates

To quantify the area savings COHESION could provide, we evaluate the on-die area costs for HW_{cc} by comparing the number of bits necessary for each COHESION. The 128 L2 caches have a capacity of 2048 lines, each resulting in 262,144 32-byte lines on-die, for a total of 8 MB. A full-map directory would require 128 bits for sharer data and 2 bits of state for each line. For COHESION, which uses a sparse directory with a set-associative structure for holding directory entries, 16 tag bits would also be required. A limited COHESION with four pointers (Dir_4B) would require 28 bits per entry for sharer state and 2 bits for coherence state. The sparse directory structure is banked to mitigate the effects port conflicts and load imbalance across L3 banks. The HW_{cc} configuration has 16k entries per bank. The overhead for a full-map directory is 9.28 MB (113% of L2) and a limited COHESION would be 2.88 MB (35.1% of L2).

Duplicate tags [64] would require 21 bits for each L2 tag. Since the directories are distributed across L3 banks, it may be necessary to replicate duplicate tag entries across banks, which leads to a 1× to 8× overhead. Duplicate tags require 736 kB * $N_{replicas}$. While a single set of duplicate tags would result in only 736 kB (8.98% of L2) overhead, the structure would still be 2048-way associative and

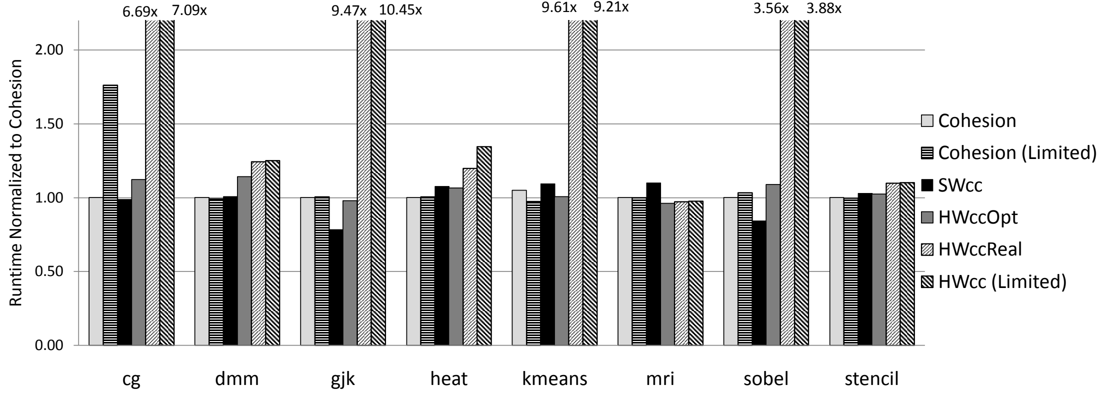


Figure 9.9: Runtime of COHESION compared to software-managed coherence (SW_{cc}) and hardware-managed coherence (HW_{cc}) using limited directories with four pointers be entry and full-map directories. We show both optimistic assumptions for HW_{cc}, which provides infinite-sized directory caches, and a large, but realizable 16k-entry directory caches.

need to service up to 32 requests per cycle on behalf of the L3 cache banks.

Clearly, none of these optimistic options is attractive for a practical design. However, COHESION has the potential to reduce hardware overhead by building smaller directories, or fewer replicas and ports for duplicate tags, compared to a pure HW_{cc} system, thus making known techniques more scalable and attractive. Our results indicate that a greater than 2× reduction in directory utilization is possible, which could yield 5% to 55% reduction in L2 overhead for directory storage on-die.

9.5.4 Application Performance

Figure 9.9 shows the runtime of our benchmarks normalized to COHESION. As the results demonstrate, two benchmarks perform better and two slightly worse with COHESION relative to SW_{cc} and optimistic HW_{cc}, while the others show insignificant differences. Compared to realistic hardware assumptions, COHESION delivers many times better performance. These benefits come from reduced message traffic (Section 9.5.1) and a reduction in the number of flush operations issued by the

SW_{cc} configurations, many of which may be unnecessary, as shown in Figure 9.2.

The negative performance difference between COHESION and optimistic HW_{cc} is less than 5% for `gjk` and `mri` and could be improved to parity with optimistic HW_{cc} using more complex coherence domain transitions not evaluated in this work. Moreover, neither benchmark is limited by coherence costs, but rather by task scheduling overhead, which is due to task granularity in the case of `gjk` and execution efficiency for `mri` because of its high arithmetic intensity. `kmeans` has a large number of atomic operations that can conflict with SW_{cc} coherence actions and lead to decreased performance due to queuing effects in the network. In some cases, HW_{cc} has the effect of dispersing the coherence actions in time, thus reducing the effect of queuing delays.

9.5.5 COHESION Summary

COHESION reduces the number of messages sent compared to a purely hardware coherent configuration, thus reducing demand on the network. COHESION is able to reduce the pressure on the directory, thus reducing the resources needed by the directory to achieve similar performance to a purely hardware coherent design. While we find an increase in the total number of messages injected when converting regions from the SW_{cc} domain to the HW_{cc} domain, we show improved performance due to the reduction in time spent issuing coherence management instructions under SW_{cc} and the timeliness of HW_{cc} messages. Overall, COHESION provides increased performance, reduced network demands, and less directory pressure for most workloads compared to SW_{cc} or HW_{cc} alone.

There is an interplay between message count, directory utilization, and execution time. Reducing the dependence on the directory by minimizing the number of lines tracked by HW_{cc} allows for fewer on-die directory resources to be provi-

sioned and can avoid pathological cases due to directory-set aliasing and capacity issues. However, doing so may lead to more messages being injected, thus reducing performance unless network capacity is increased. The trade-off is the level of achieved performance and the amount of programming effort beyond developing an application using HW_{cc} alone that is applied to separate data accesses into SW_{cc} and HW_{cc} domains.

9.6 Summary and Discussion

In this chapter we evaluate the use of different software and hardware coherence techniques. We show the scalability of a software-managed coherence scheme that we call the Task-Centric Memory Model. We show that it can achieve performance that approaches omniscient coherence implementations and performs slightly better than a highly-optimistic hardware coherence implementation. The evaluation of scalable probe filtering indicates that probe filtering, a common technique in contemporary CMPs, can be scaled if support is available for accelerating broadcast and collective messages. However, when the number of cores grows beyond 128 in our evaluation, SPF fails to scale. We next evaluate WayPoint, a hardware coherence mechanism based on a subset sparse directory resident on-die. WayPoint can provide additional capacity and associativity to the directories dynamically. We find that WayPoint can achieve good performance with greatly reduced area compared to a conventional sparse directory. Lastly, we evaluate Cohesion, a technique for combining both our software coherence techniques and a simple hardware coherence implementation. We find that Cohesion can reduce network traffic and directory resource demands by roughly a factor of two. These benefits can be translated into reduced hardware costs without compromising performance.

CHAPTER 10

Related Work

In this section we discuss previous work related to the Task-Centric Memory Model, WayPoint, and Cohesion.

10.1 Task-Centric Memory Model Related Work

Compute accelerators have developed an array of memory models that emphasize compute density and parallel scalability. Accelerators have leveraged the lack of legacy software constraining their design and the high degree of parallelism inherent in their workloads. By design, existing software APIs and programming models for accelerator systems are parallel by design with very weak memory models and implied coherence guarantees. On the other hand, software APIs and parallel programming models for coherent CMPs have strong consistency models and hardware coherence. This section discusses existing programming models and programming languages used by compute accelerators and parallel coherent CMPs. We illustrate how these models exploit characteristics of their workloads and the underlying architecture to achieve better performance or to enhance programmability.

Many of the prevalent models for accelerators exploit the existence of coarse-grained synchronization and the relative lack of fine-grained sharing in workloads. Accelerators have achieved success relying on software for handling coherence actions and allow relaxed memory orderings, thus aiding hardware scalability and

improving power and performance density. Here we survey memory models and programming models for parallel systems and compare the models with the approach presented in Chapter 5. With the increased interest in accelerator platforms such as GPUs for general-purpose computation, we see an opportunity for memory models that are less reliant upon hardware to become widespread as core counts continue to rise and the distinction between CMP and accelerator begins to blur.

The Task-Centric Memory Model targets systems with a single address space and hardware-managed caches without hardware-managed coherence. Our approach contrasts with that of existing accelerators using software-managed scratchpads [4,57] or designs more similar to contemporary CMPs, where caches are kept coherent transparently to software [14]. Friedman [111] discusses a form of hybrid coherence where there are strong and weak memory operations. Friedman goes on to construct a memory model that eliminates latencies that are unavoidable with a conventional memory model. The model is similar to what is supported on commercially available multicore processors today where synchronization operations are done using a set of atomic operations that adhere to stricter ordering rules than normal programmed loads and stores. Moreover, the use of strong and weak operations is very similar to the local versus global distinction used in Rigel for memory operations.

Leverich et al. [112] investigate the implications of choosing between two different memory system configurations, hardware-coherent caches and software-managed scratchpads, for future CMPs and demonstrate that software coherence actions can provide benefit to cached systems. A third choice not investigated in that work, *incoherent software-based* architectures, is most similar to the Task-Centric Memory Model. Furthermore, prototype systems with hardware caches, but without hardware coherence, such as CEDAR [54], have been built. These

same techniques are being reapplied to accelerator systems today, such as the Rigel accelerator [42] used as the basis for this work.

10.1.1 Parallel Programming Models

Many parallel models for existing CMPs, such as Intel's Threading Building Blocks (TBB) [79] and Cilk [80], use explicit task generation. Models such as OpenMP [113] use implicit task generation. Explicit task generation is also used in the Rigel Task Model, but we limit RTM to BSP semantics while TBB also supports fork-join parallelism. TBB and Cilk allow for interactions between tasks and make use of parent-child communication through shared memory, which relies upon the existence of a coherent address space.

Underlying many of the models used by accelerators is the bulk-synchronous parallel model (BSP) [60]. BSP continues to be reflected in accelerator languages prevalent today, including CUDA [114] from NVIDIA and OpenCL [30]. CUDA and OpenCL are used to map data-parallel kernels to highly parallel systems comprising possibly hundreds of processing elements in a bulk-synchronous fashion.

While CMPs continue to support unrestricted sharing patterns and accelerators usually adopt shared-nothing programing models, we see a potential for an intermediate design point, such as the Task-Centric Memory Model, that exploits the structure of accelerator applications by using a software-protocol and minimal hardware to provide the programmability afforded by CMPs while achieving the scalability of accelerators.

10.1.2 Parallel Memory Models

The Rigel memory model and coherence mechanisms are akin to software coherence mechanisms used to provide the illusion of a single address space for

distributed shared memory (DSM) systems [102, 115]. Two DSMs, Midway [116] and Munin [103], used flexible consistency models to achieve parallel scalability. Midway allowed for a high degree of latency tolerance by associating individual data items with synchronization operations and only guaranteeing that the data was visible after acquiring the associated synchronization object. The system also supported multiple consistency models concurrently in one program. Munin was based on data types specified by the programmer that allowed for communication-based per-type optimizations to be exploited by the runtime. Munin and Midway are analogous to a software-only hybrid memory model.

The consistency guarantees we investigate for write-output data at RTM task boundaries are similar to Scope Consistency [117] in that dirty data is implicitly made coherent at the end of the task's scope and updates can be deferred until the scope is reopened. Reopened in the case of RTM means starting a new task or interval following a barrier. The Cooperative Shared Memory model [71] provides a similar model to Rigel. Cooperative Shared Memory relies on software to properly label shared accesses for performance and achieves scalable performance using a reduced complexity hardware coherence protocol (Dir_1SW).

The BSP model was described by Valiant [60]. BSP continues to be reflected in languages prevalent today including CUDA [114] from NVIDIA and OpenCL [30]. As mentioned previously, CUDA is used to map data-parallel kernels to GPUs comprising hundreds of processing elements in a bulk-synchronous fashion, but requires SIMD-friendly code to achieve high execution efficiency [12]. DeNovo [118] is an attempt to exploit race-free and deterministic software to build an architecture that reduces the strain on hardware coherence mechanisms. The work is similar to the TCMM because it exploits program structure, in the case of DeNovo race freedom, to relax the constraints placed on hardware.

10.1.3 Accelerator Workloads

Examples of data- and task-parallel workloads that motivate our investigation of a task-parallel model include recognition, mining, and synthesis (RMS) [81] and physical simulation applications [15] for providing more realistic virtual worlds that are being investigated by Intel. A later study from Intel [119] comparing the performance of GPUs and CPUs for throughput-oriented workloads uses benchmarks similar to the workloads evaluated here.

A variety of highly-parallel workloads, such as the PARSEC [98] and ALP-Bench [120] suites, have been evaluated for conventional multicore processors. Accelerator workloads targeting current-generation GPUs have been studied [19], while studies motivating future accelerator architectures have focused on characterizing visual computing workloads [12]. While these studies investigate the scalability of visual computing workloads, we go further to point out the sharing patterns relevant to coherence management and show how these characteristics can be exploited in the design of future compute accelerators.

10.2 WayPoint Related Work

In this section we discuss previous work related to scalable cache coherence and cache associativity.

10.2.1 Coherence Management

Two classes of protocols used for maintaining cache coherence are directories [67] and snooping or broadcast schemes [61]. On the one hand, the scalability of simple directory protocols is limited by directory storage overhead, which grows with the square of the number of processors, and contention at the directory [121]. Limited

schemes such as coarse vectors [65], which reduce the size of the sharing vectors, still have storage overhead that grows super-linearly with the amount of memory being tracked. On the other hand, broadcast protocols become limited at high core counts by bandwidth requirements and ordering constraints placed on the interconnect.

Cooperative Shared Memory and the underlying Dir_1SW protocol [71] use programmer annotations to reduce hardware complexity, achieving good performance when the software protocol is respected and correctness even when it is not. The LimitLESS directory described by Chaiken et al. [70] maintains a small list of sharers in hardware and faults to software when the list overflows. The protocol suffers from a high number of faults if the number of sharers is high or unstable. WayPoint also uses a fallback mechanism to handle uncommon cases in directory coherence schemes, but, unlike LimitLESS and Dir_1SW, WayPoint addresses directory cache associativity and capacity overflow rather than sharer list overflow. Furthermore, the fallback mechanism for WayPoint is designed to be implemented as part of the on-die coherence logic, obviating the need for software intervention—a potentially costly operation if cores are far from the directory.

Michael and Nanda [69] evaluate the use of a set-associative directory cache at each of 16 SMP nodes. WayPoint targets workloads with different data sharing characteristics, using the storage and communication cost models inherent to the single-chip designs, demonstrating the growing problem of directory associativity for CMPs. Furthermore, we investigate a novel scheme that uses linked lists of sharer data, whereas Michael and Nanda do not discuss their implementation of in-memory directories.

Scalable Cache Interface [122] (SCI) is a sparse directory scheme based on linked lists of sharers used to minimize storage overhead. SCI reduces overhead

by only tracking those lines currently cached and reduces contention by spreading sharing data across caches. While both SCI and WayPoint use linked list structures, SCI differs from WayPoint in that it keeps lists of *sharers* in *all* cases, whereas WayPoint has lists of *directory entries* only in the case of directory cache overflow. The use of linked lists of directory entries simplifies the design relative to one that uses linked lists of sharers, since transactions no longer span multiple cache banks, multiple sharer updates, and multiple interacting cache controllers, which makes serialization and the atomicity of coherence transactions easier to guarantee.

Recent work investigating scalable coherence includes Token Coherence [123], which associates a number of tokens with any given line. While not requiring an ordered network, as in snoopy protocols, nor requiring large centralized structures, as in conventional directory schemes, Token Coherence still relies upon broadcasts to find tokens and requires all sharing state to be resident on-chip. WayPoint allows for even less on-die storage due to its ability to flush noncritical directory entries to memory. Issues in the initial Token Coherence work were addressed by adding a directory to the token protocol [124]. However, this later work does not reduce the overhead of on-die directories, nor does it deal with the associativity problem addressed by WayPoint.

Many of these schemes are predicated on the empirical observations that there are generally few concurrent sharers of any given line and that data migrates between processors over time [125]. However, the SPLASH [126], transaction processing, and database applications that have been used to evaluate previous designs have different sharing patterns from those applications used to evaluate future 1000-core CMPs. The CMP applications tend to have a high degree of read sharing [12,15,98]. Furthermore, these applications have less migratory data due to the finer thread granularity and the adoption of task-based programming

models, such as NVIDIA's CUDA [114] and Intel's Threading Building Blocks [79], and instead have a greater degree of producer-consumer sharing between tasks [12, 88]. Further note that these tasks are often not required to be concurrently running, but rather perform consumer-producer communication across the parent-child spawn point or, in BSP, across barriers.

10.2.2 Directory Cache Associativity

Qureshi et al. [127] present the V-way cache, which decouples data-store sizing from tag store sizing by implementing an extra level of pointer indirection on a tag access. The effective associativity provided by the scheme scales with the tag store associativity, which is still constrained by area, power, and latency considerations. A victim cache [128] also provides a way to mitigate the effects of limited associativity by providing a fully-associative structure that can capture entries that suffer conflict evictions while still being actively used. However, neither scheme addresses capacity misses, nor can they compensate for highly imbalanced sets. The inability to address limited capacity and set imbalance stems from the fact that either scheme would be limited in capacity by what can be provided on-chip; WayPoint's ability to track entries is not limited by on-die resources.

Caches with programmable hashing functions, such as the PADded Cache [129] or the Balanced Cache [130], are potential solutions to the associativity problem, but also suffer from the limits of on-die data store capacity. WayPoint alleviates this limitation by using cacheable memory to store those directory entries which cause conflict or capacity misses in the directory caches. Kharbutli et al. [131] use specially constructed hash functions based on prime numbers to index into tag arrays, preventing conflicts caused by associativity demand imbalance with traditional hash functions. However, doing so lengthens the critical path for cache

accesses by adding extra stages in the directory cache lookup to evaluate the hash functions and the scheme does not address capacity misses.

The Indirect Index Cache [132] increases the effective associativity of last-level uniprocessor caches by augmenting a small set-associative primary structure with a secondary SRAM structure which contains linked lists of conflicting entries from primary structure sets. WayPoint stores conflicting directory cache entries in cached memory instead of SRAM, reducing on-die storage requirements and enabling greater scalability. While storing data cache entries in DRAM would be counterproductive, doing so for directory cache entries is profitable due to the higher cost of evictions.

10.3 Cohesion Related Work

In this section, we put Cohesion into context with previous work in scalable hardware-managed and software-managed coherence. Many of the previous coherence techniques could be adapted to Cohesion, replacing the baseline HW_{cc} and SW_{cc} protocols in our implementation. Therefore, many of the schemes can be viewed as *complementary* approaches to reducing the cost of coherence management. Although these protocols could be adapted to Cohesion, previous work mostly focuses on multi-socket coherence, whereas our work focuses solely on chip multiprocessor coherence, thus resulting in different design constraints for which we must account. In our context, the memory and memory bandwidth does not scale up as more cores are added, as is true with multi-socket systems. Moreover, the latencies involved in on-die communication may be much less than those expected across sockets.

10.3.1 Hardware Schemes

Snoopy coherence protocols [61] rely upon ordered networks and broadcasts to maintain consistency across caches, but are difficult to scale. Directory schemes [67] provide a scalable alternative, but full implementations suffer from high memory overhead. Limited directories [99] and sparse directories [65] can aid in reducing memory overhead with the potential cost of extra messages.

Marty and Hill [68] leverage the virtualized nature of server consolidation workloads to reduce message overhead, but do not focus on highly parallel visual computing and high-performance workloads. Moreover, Marty and Hill use a full backing store. A full backing store has high memory requirements, while our designs use sparse directories that only track a subset of cache lines resident on-die. Coarse-grain coherence tracking [62] and RegionScout [63] both propose mechanisms to reduce coherence traffic in broadcast-based systems by managing coherence at a coarser granularity than a line. While these techniques can reduce storage costs, both mechanisms impose restrictions on alignment and sizing of coherence regions and may lead to increased message traffic; both are situations we wish to avoid with Cohesion.

Modern accelerator hardware, such as IBM's Cell [57] and NVIDIA's Tesla [4], provide a variety of access modes to data located in different physical address spaces, but require explicit software management of data movement between the various memories. Leverich et al. [112] demonstrate the benefit of software management of data movement on hardware cached platforms, which Cohesion can facilitate with the added precision of variable coherence regions. Moreover, Cohesion can provide additional benefit by cutting across the quadrants used in the taxonomy of Leverich et al., thus providing the benefits of two or more models without the limitations cited in their work.

10.3.2 Software-Based and Hybrid Schemes

Distributed shared memory (DSM) provides the illusion of a single coherent address space across processors at a coarse grain using virtual memory and a runtime system, in the case of TreadMarks [102], and at a fine grain using compiler support such as in Shasta [115]. While these approaches could trivially support incoherence due to their distributed memory architecture, they synthesized coherence when needed fully in software. Cooperative Shared Memory [71] uses software hints to reduce the complexity and cost of hardware-supported coherence. Software-assisted hardware schemes, such as LimitLESS [70], trap to software when the number of sharers supported by hardware is exceeded. CSM and LimitLESS suffer from high round-trip latency between directory and cores in a hierarchically cached system, and require all data to be tracked by the coherence protocol, resulting in unnecessary traffic for some data.

Previous work on distributed memory multiprocessors investigated hybrid schemes that combine message passing with hardware-managed coherent shared memory. FLASH [133] and Typhoon [134] utilized programmable protocol controllers that support customized protocols. User-Level Shared Memory in Typhoon made fine-grained access control a key component of customizing protocols. Cohesion provides a mechanism to allow such customization without a separate protocol controller. Munin [103] used parallel program access patterns to provide different consistency guarantees to different classes of data. Multiphase Shared Arrays [135] provide a means for the programmer to specify access modes for array data and to change the mode for different program phases. Cohesion on an integrated shared memory multiprocessor captures these features with modest hardware and an intuitive programming model that does not require a message passing component. Moreover, Multiphase Shared Arrays could be adopted to

a system that supports Cohesion whereby coherence domain transitions would coincide with phase changes in the Multiphase Shared Array model.

The ability to change the strictness of regions of memory is available in a limited form on x86 processors with write combining [8], and PowerPC allows for pages to be set incoherent [136]. Both mechanisms work at page granularity and require operating system support. A hybrid x86 system with a consistent programming model for GPU and CPU cores using a subset of the address space has been proposed by Saha et al. [137]. Unlike Cohesion, their work does not investigate dynamic transitions between different coherence domains. Furthermore, while their work focuses solely on a host-accelerator model with one CPU core, Cohesion is demonstrated using 1024 cooperating cores.

WildFire [138] used operating system support to transition between line-level ccNUMA shared memory and a form of COMA known as Coherent Memory Replication (CMR). CMR and ccNUMA pages trade off coherence space overhead for coherence message overhead since all lines in the system were tracked by hardware and CMR pages require replication of coherence state. Copy operations are required by replication under CMR, which are eliminated by Cohesion. Unlike WildFire, Cohesion provides symmetric access to the last-level cache, analogous to memory in DSM systems, and does not present a trade off between state and message overheads; it reduces both when using SW_{cc}.

Reactive NUCA, described by Hardavellas et al. [139], uses operating system remapping of page-sized regions on a distributed NUCA multicore. Hardavellas et al. show that different regions of memory possess different coherence needs, which offers an opportunity for hybrid coherence, and many of the trade-offs in scaling distributed versus shared cache architectures. In contrast to Cohesion, their work evaluates mostly server and multiprogrammed workloads scaling to eight or 16 cores while we target the class of visual computing applications that

tend to have higher degrees of read sharing and a higher degree of structure in their sharing patterns [12, 15, 88, 98] and are shown to scale to 100+ cores.

CHAPTER 11

Summary and Conclusions

This dissertation demonstrates the feasibility and the necessity of a hybrid memory model for 1000-core microprocessors. We present results for a software-managed coherence scheme, a lightweight directory-based hardware coherence protocol, and a mechanism for allowing on-the-fly coherence domain transitions between these two coherence protocols. While we have shown the viability of our approach on a symmetric 1024-core accelerator, we believe that there is an opportunity to extend this work to heterogeneous platforms incorporating both general-purpose, ILP-oriented cores and accelerator-like, throughput-oriented cores. A hybrid memory model would allow these systems to achieve better performance and power levels, while also enabling a more conventional programming abstraction for developers.

In this chapter we discuss the main conclusions of our work. We provide a general perspective on the design of memory models for accelerators, we discuss the observed benefits of using a mix of coherence protocols in a single processor design, we present an argument for different levels of symmetry for different aspects of processor design. We close with a brief summary of the work presented in this dissertation.

11.1 Implementing Coherence in Future Processors

This dissertation demonstrates that a 1000-core processor that exports a coherent memory model to programmers is feasible. We have presented the trade-offs and evaluated many of the costs that a 1000-core CMP architect must consider. We have not defined exactly which choices in the space of hybrid architectures is appropriate. What will determine the proper set of choices are within the space of hybrid memory models the emerging applications—many of which have yet to be developed.

What is clear is that currently implemented protocols for hardware cache coherence are inappropriate for 1000-core processors due to their high overhead. We have evaluated scalable probe filtering and full directories. We find that these approaches either lack scalability or have prohibitively high implementation costs. We have provided a set of protocols and mechanisms for wedding diverse coherence mechanisms on a single processor. We believe hybrid memory models that mix aspects of software-managed and hardware-managed coherent designs will be necessary for future, highly parallel architectures.

11.2 Symmetry Versus Asymmetry

A trade-off at the heart of this dissertation is the mix of symmetry and asymmetry in processor design and how it will evolve over the coming generations of CMP. The choice of symmetry over asymmetry, or homogeneity over heterogeneity, is not binary for an entire chip design, but rather is a choice that must be made at a variety of levels within the design. Choices include level of microarchitectural complexity [140], type of ISA, type and depth of cache hierarchy, the number of vector units, a SIMD configuration or MIMD configuration (or both [141]) configurations, and, as we discuss in this work, levels of support for accessing a

shared address space.

This dissertation takes the view that symmetry is necessary at the memory model provided to the developer to achieve programmability and portability, whereas asymmetry should be employed to achieve the best power and area efficiency for compute resources. We list here the key issues related to symmetry in processor design.

- **The value of asymmetry**. Having cores of different complexities and performance characteristics is ideal for exploiting the mix of extreme data parallelism and the inherent sequential execution present in real applications [38]. As demonstrated by SoC designs used in embedded devices, asymmetry can be leveraged to build lower-power devices by incorporating efficient accelerators and shifting compute from high-speed, yet inefficient, processing cores when timing slack exists. Asymmetry allows the system to provide the best mix of area, power, and performance across a broad range of workloads.

- **The cost of asymmetry.** Asymmetry leads to extra degrees of freedom in the design process, which in turn adds to the complexity borne by the software developer. If the asymmetry is not properly abstracted, the developer must decide up-front to use one resource or the other, or pay the cost of developing for both. Otherwise, the chip architect has to build and verify two mechanisms and make sure that they compose properly. Designing two mechanisms to achieve a single goal leads to opportunity costs reducing the efficacy of both. One example of this was the T3D, a commercial system, where the designers implemented three ways to access memory [50]. In retrospect, the designers found this burdened the compiler and the developer with the task of deciding how to access memory without having the ability

to make an informed decision. Design teams have fixed resources. Should the decision be made to design three mechanisms in place of one, the added design effort for the two additional mechanisms comes at the cost of greater optimization for the one. Moreover, in the initial design phase, it can be hard to choose the right mix of components to match the broad spectrum of applications the software developers will want to run on the asymmetric system.

- **The value of symmetry.** Design costs of a symmetric design are lower than an asymmetric design since only one piece is replicated. Moreover, optimization efforts in both hardware and software can focus on targeting the sole resource in the design. The software abstraction that targets the symmetric processor need not compromise the collection of processing resources supported by an asymmetric design. Scheduling complexity in the runtime or operating system is reduced, since there is only one resource type to target.

- **The cost of symmetry.** Using only a single resource is not globally optimal across multiple workloads. A core that matches a particular workload, or even a class of workloads, will be sub-optimal for the remaining workloads. General-purpose processors today have to cater to server-class workloads that may be throughput-oriented and memory-bound, HPC applications that may be compute-bound and latency intolerant, and mobile applications where parsimonious use of energy and fast response times for user requests are critical design goals. Optimization efforts may in fact be higher in the symmetric case as developers try to map applications to inappropriate hardware resources. As an example, attempting to map a highly data-parallel application with a high degree of divergence between threads to a

SIMD-only architecture may lead to much greater programming effort than using a mix of MIMD and SIMD cores.

These issues motivate a design with as much asymmetry as is necessary, but no more. Asymmetry can enable hardware optimized for a particular workload, but it places the onus on software to make proper partitioning, data marshalling, and scheduling decisions. Symmetry, on the other hand, can reduce design times and programmer effort, but at the cost of hardware inefficiencies.

Current general-purpose systems are symmetric with each core having the same complement of features, the same ISA, and the same shared view of memory. Future systems will have more asymmetries. At the time of this writing, the major CPU and GPU manufacturers are presently shipping or about to ship processors that incorporate both a CPU component and a GPU component. These first generation hybrid systems will have asymmetry at the core level and at the memory level, with each cluster of cores maintaining independent cache and memory resources.

This dissertation has made the case for providing a consistent, or symmetric, memory model to developers by masking the inconsistencies, or asymmetries, in hardware and runtime as much as possible while still enabling asymmetries in the hardware to achieve scalability. The goal is to take advantage of the two extremes of processor core: general-purpose ILP-centric processors tailored to sequential workloads and special-purpose accelerator cores that deliver area and power efficiency for parallel workloads. While we have not evaluated a heterogeneous platform directly, we note that general-purpose systems that are cache coherent and provide a shared single address space to software exist. We provide the components that enable accelerators with weaker hardware-defined memory models to plug into that model in a consistent fashion.

We see the memory model abstraction as the *lingua franca* of parallel com-

puter systems that can allow disparity in cores to be masked. For these systems to be useful, the accelerator cores must target a large-enough class of workloads to make the opportunity cost of choosing to have fewer general-purpose cores on-die minimal. To achieve this end, we present the Rigel architecture inspired by the design elements we see as necessary for a scalable, programmable accelerator architecture. Likewise, we we must have the ability to achieve reasonable sequential performance if we are to overcome Amdahl's law for the applications for which Gustafson's law does not fully apply.

11.3 Summary

We now summarize this dissertation. We break the work down into three components (1) The motivation for and design of a software-managed memory model based on task-based parallelization of parallel workloads that synchronize using barriers, (2) the development of a lightweight directory-based cache coherence protocol that achieves low hardware overhead while avoiding the performance pathologies of other minimalistic approaches to hardware cache coherence, and (3) a method for incorporating both a software and a hardware coherence protocol under a single memory model.

11.3.1 The Task-Centric Memory Model

We present a mechanism for providing the appearance of a shared single address space cache hierarchy on a system with hardware caches, but not hardware coherence. Analysis of scalable workloads shows a distinct pattern of sharing when programmed using a prevalent programming model that includes barriers for synchronization and task queues for work distribution. These patterns can be exploited in building a software-managed coherence protocol. Our efforts have

shown that a particular software-managed coherence protocol, the Task-Centric Memory Model, is a feasible design for these workloads. However, we must consider the scheduling policy of invalidations and writeback operations to achieve good performance. Due to the conservative nature of a software-managed approach to coherence, we find many of the software coherence actions are performed on data not present in the cache, representing a loss of execution stream and network efficiency. We also see the need for more tools and additional architecture support for the Task-Centric Memory Model to make the process of determining what date to flush and when to flush the data less of a burden for programmers and to make the process more efficient at runtime.

11.3.2 WayPoint: Scalable Hardware Cache Coherence

We present WayPoint, a lightweight coherence scheme that makes use of small, on-die caches of directory entries and in-memory lists of directory entries that overflow the on-die directory cache. The key idea is that a small subset of sharer information must be accessible with low latency while nearly all shared data must be tracked in some fashion to avoid performance cliffs. We find that WayPoint can achieve scalable performance with little overhead; furthermore, it serves as the hardware dual of the Task-Centric Memory Model in our proposed hybrid memory model. While WayPoint removes performance loss due to invalidations and can reduce on-die resources considerably, it still requires additional message traffic and may compete with demand accesses to the last-level cache and memory. Finally, we find that for each of our workloads there is a minimum directory cache size or associativity below which performance degrades greatly due to thrashing. However, the performance cliff is at a much reduced point compared to sparse directory protocols without WayPoint.

11.3.3 Cohesion: A Hybrid Memory Model for Accelerators

Our initial effort to construct a hybrid memory model, which we call Cohesion, provides a proof-of-principle result. Our results show that hybrid coherence can reduce the on-die directory needs and reduce the demands placed on the network relative to a design without Cohesion. Cohesion provides a knob that developers can turn allowing them to trade off programming effort for performance. Further, work remains to be done optimizing Cohesion to take advantage of common sharing patterns at coherence domain transitions.

11.4 Conclusions

This dissertation demonstrates the feasibility of a hybrid memory model and shows that future multicore processors must adopt a hybrid memory model if they are to continue to scale core counts. Furthermore, we propose using a hybrid memory model as the means to synthesize future heterogeneous systems that will incorporate simple accelerator cores optimized for throughput and larger ILP cores optimized for latency reduction.

We conclude with two general observations. First, software-managed coherence protocols provide the advantage of executing an arbitrary number of invalidations with a single non-local communication. Our software-managed protocol, the Task-Centric Memory Model, accomplishes low-overhead coherence actions with distributed invalidation operations, writes that do not require any coherence actions, and a barrier operation to synchronize. The appearance of a shared single address space is achieved with near-zero fixed hardware cost. Second, we find that hardware-managed coherence has the advantage of supporting task migration, task-stealing, and optimistic privatization of potentially shared data. By integrating the two coherence mechanisms in the same design, we achieve accelerator-like

scalability with a general-purpose programming abstraction.

APPENDIX A

Task-Centric Memory Model Formal Specification

This appendix presents a TLA+ [142] specification for the Task-Centric Memory Model. The specification has been used to verify basic properties of the system, including liveness and deadlock freedom. More details on the implementation of the TCMM are available in Chapter 5

```
1  ---------------- MODULE SWCoherence ----------------
2  EXTENDS TLC, Naturals, FiniteSets
3  CONSTANT Addr, TaskID, OpType, BlockState
4  CONSTANT MaxTasksPerInterval, MaxIntervals
5  VARIABLE tasks, memstate, inbarrier
6  VARIABLE accessesPerTask, intervalsCompleted
7  ----------------------------------------------------
8  Init ≜
9     ∧ tasks = [addr ∈ Addr ↦ {}]
10    ∧ memstate = [addr ∈ Addr ↦ "clean"]
11    ∧ inbarrier = {}
12    ∧ accessesPerTask = [task ∈ TaskID ↦ 0]
13    ∧ intervalsCompleted = 0

15 TypeInvariant ≜
16    ∧ tasks ∈ [Addr → SUBSET TaskID]
17    ∧ memstate ∈ [Addr → BlockState]
18    ∧ accessesPerTask ∈ [TaskID → Nat]
19    ∧ intervalsCompleted ∈ Nat

21  Set the list of permutation that are allowable for constants so that TLC can
22  take advatnage of symmetries in the state space.
23 TaskPerms ≜ Permutations(TaskID)
24 ----------------------------------------------------
25 Constraint ≜ intervalsCompleted ≤ MaxIntervals
26 ----------------------------------------------------
27  The Allow ∗ () operations are used to inhibit transitions when a block is not
28  held in the proper state or the task is in a barrier
29 AllowGlobal(tid, addr) ≜
30    ∧ tid ∉ inbarrier
31    ∧ ((memstate[addr] ∈ {"clean"})
32          ∨ ((tid ∈ tasks[addr]) ∧ (memstate[addr] ∈ {"globally_coherent"})))

34 AllowPrivLocalStore(tid, addr) ≜
35    ∧ tid ∉ inbarrier
36    ∧ (memstate[addr] ∈ {"clean"}
37         ∨ ((memstate[addr] = "private_clean" ∨ memstate[addr] = "private_dirty") ∧
38             (tasks[addr] = {tid})))

40 AllowPrivLocalLoad(tid, addr) ≜
41    ∧ tid ∉ inbarrier
42    ∧  (memstate[addr] ∈ {"clean"}
43          ∨ ((memstate[addr] = "private_clean" ∨ memstate[addr] = "private_dirty") ∧
44              (tasks[addr] = {tid})))

46 AllowImmLocalLoad(tid, addr) ≜
```

```
47      ∧ tid ∉ inbarrier
48      ∧ ((memstate[addr] ∈ {"clean"})
49            ∨ ((tid ∈ tasks[addr]) ∧ (memstate[addr] ∈ {"immutable"})))

51   AllowInvalidate(tid, addr) ≜
52      ∧ tid ∉ inbarrier
53      ∧ memstate[addr] ∈ {"private_clean", "immutable"}
54      ∧ tid ∈ tasks[addr]

56   AllowWriteback(tid, addr) ≜
57      ∧ tid ∉ inbarrier
58      ∧ memstate[addr] ∈ {"private_dirty"}
59      ∧ tid ∈ tasks[addr]

61   |-----------------------------------------------------------------------|
62    ForceClean(): Make all lines clean and unowned.
63   ForceClean ≜
64      ∧ tasks' = [addr ∈ Addr ↦ {}]
65      ∧ memstate' = [addr ∈ Addr ↦ "clean"]

67    EnterBarrier(): Conditions under which a task enters a barrier and stops
68    executing memory operations.
69   EnterBarrierNone(tid) ≜
70       Once maximum tasks are reached, enter the barrier.
71      ∧ accessesPerTask[tid] = MaxTasksPerInterval
72      ∧ inbarrier' = inbarrier ∪ {tid}
73      ∧ UNCHANGED ⟨tasks, memstate, accessesPerTask, intervalsCompleted⟩

75   EnterBarrierEager(tid) ≜
76       Once maximum tasks are reached, enter the barrier.
77      ∧ accessesPerTask[tid] = MaxTasksPerInterval
78      ∧ memstate' = [addr ∈ Addr ↦
79          IF (tid ∈ tasks[addr] ∧ Cardinality(tasks[addr]) = 1)
80           THEN "clean"
81           ELSE  memstate[addr]]
82      ∧ tasks' = [addr ∈ Addr ↦ tasks[addr] \ {tid}]
83      ∧ inbarrier' = inbarrier ∪ {tid}
84      ∧ UNCHANGED ⟨accessesPerTask, intervalsCompleted⟩

86    LimitAccesses(): Internal call to limit the state space of the model.
87    Otherwise, tasks can execute as many tasks per interval as they want.
88   LimitAccesses(tid) ≜
89       Only allow accesses when not in the barrier
90      ∧ ¬(tid ∈ inbarrier)
91       Only allow access if the maximum has not yet been reached.
92      ∧ accessesPerTask[tid] < MaxTasksPerInterval
93       Once entering the barrier, reset the count of accesses per interval per task
```

```
94    ∧ accessesPerTask' = [accessesPerTask EXCEPT ![tid] = accessesPerTask[tid] + 1]

96   DoBarrier(): All tasks are waiting on the barrier. Reset barrier. and move
97   on to the next interval. TODO: We could force it to say all tasks must make
98   all data clean before entering the barrier.
99  DoBarrierNone ≜
100     When all tasks are waiting on the barrier, the set of tasks in the barrier
101     becomes the empty set.
102   ∧ Cardinality(TaskID) = Cardinality(inbarrier)
103   ∧ inbarrier' = {}
104   ∧ accessesPerTask' = [tid ∈ TaskID ↦ 0]
105   ∧ intervalsCompleted' = intervalsCompleted + 1
106   ∧ UNCHANGED ⟨tasks, memstate⟩

108 DoBarrierLazy ≜
109     When all tasks are waiting on the barrier, the set of tasks in the barrier
110     becomes the empty set.
111   ∧ Cardinality(TaskID) = Cardinality(inbarrier)
112   ∧ inbarrier' = {}
113   ∧ accessesPerTask' = [tid ∈ TaskID ↦ 0]
114   ∧ intervalsCompleted' = intervalsCompleted + 1
115   ∧ ForceClean

117 CoherenceLazy(tid) ≜ DoBarrierLazy ∨ EnterBarrierNone(tid)
118 CoherenceEager(tid) ≜ DoBarrierNone ∨ EnterBarrierEager(tid)
119 ├──────────────────────────────────────────────────────────────────────┤
120 LLDPrivBlock(tid, addr) ≜
121   ∧ AllowPrivLocalLoad(tid, addr)
122   ∧ LimitAccesses(tid)
123   ∧ tasks' = [tasks EXCEPT ![addr] = @ ∪ {tid}]
124   ∧ memstate' = [memstate EXCEPT ![addr] =
125       IF  @ = "clean"              THEN "private_clean"  ELSE
126       IF  @ = "private_clean"      THEN "private_clean"  ELSE
127       IF  @ = "private_dirty"      THEN "private_dirty"  ELSE
128       IF  @ = "immutable"          THEN "ERROR_STATE"    ELSE
129       IF  @ = "globally_coherent"  THEN "ERROR_STATE"    ELSE  "ERROR_STATE" ]
130   ∧ UNCHANGED ⟨inbarrier, intervalsCompleted⟩

132 LSTPrivBlock(tid, addr) ≜
133   ∧ AllowPrivLocalStore(tid, addr)
134   ∧ LimitAccesses(tid)
135   ∧ tasks' = [tasks EXCEPT ![addr] = @ ∪ {tid}]
136   ∧ memstate' = [memstate EXCEPT ![addr] =
137       IF  @ = "clean"              THEN "private_dirty"  ELSE
138       IF  @ = "private_clean"      THEN "private_dirty"  ELSE
139       IF  @ = "private_dirty"      THEN "private_dirty"  ELSE
```

```
140         IF   @ = "immutable"          THEN "ERROR_STATE" ELSE
141         IF   @ = "globally_coherent"  THEN "ERROR_STATE" ELSE  "ERROR_STATE" ]
142    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted⟩

144  LLDImmuteBlock(tid, addr) ≜
145    ∧ AllowImmLocalLoad(tid, addr)
146    ∧ LimitAccesses(tid)
147    ∧ tasks' = [tasks EXCEPT ![addr] = @ ∪ {tid}]
148    ∧ memstate' = [memstate EXCEPT ![addr] =
149         IF   @ = "clean"              THEN "immutable"       ELSE
150         IF   @ = "private_clean"      THEN "ERROR_STATE"     ELSE
151         IF   @ = "private_dirty"      THEN "ERROR_STATE"     ELSE
152         IF   @ = "immutable"          THEN "immutable"       ELSE
153         IF   @ = "globally_coherent"  THEN "ERROR_STATE"     ELSE  "ERROR_STATE" ]
154    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted⟩

156  InvalidateBlock(tid, addr) ≜
157    ∧ AllowInvalidate(tid, addr)
158    ∧ LimitAccesses(tid)
159    ∧ tasks' = [tasks EXCEPT ![addr] = @ \ {tid}]
160    ∧ memstate' = [memstate EXCEPT ![addr] =
161         IF   @ = "clean"              THEN "ERROR_STATE"     ELSE
162         IF   @ = "private_clean"      THEN "clean"           ELSE
163         IF   @ = "private_dirty"      THEN "ERROR_STATE"     ELSE
164         IF   @ = "immutable"          THEN
165           IF Cardinality(tasks[addr]) = 1 THEN "clean" ELSE  "immutable" ELSE
166         IF   @ = "globally_coherent"  THEN "ERROR_STATE"     ELSE  "ERROR_STATE" ]
167    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted⟩

169  WritebackBlock(tid, addr) ≜
170    ∧ AllowWriteback(tid, addr)
171    ∧ LimitAccesses(tid)
172    ∧ tasks' = IF memstate[addr] = "private_dirty"
173                 THEN tasks   WB to dirty line is still owned
174                 ELSE  [tasks EXCEPT ![addr] = @ \ {tid}]
175    ∧ memstate' = [memstate EXCEPT ![addr] =
176         IF   @ = "clean"              THEN "ERROR_STATE" ELSE
177         IF   @ = "private_clean"      THEN "ERROR_STATE" ELSE
178         IF   @ = "private_dirty"      THEN "private_clean"   ELSE
179         IF   @ = "immutable"          THEN "ERROR_STATE" ELSE
180         IF   @ = "globally_coherent"  THEN "ERROR_STATE" ELSE  "ERROR_STATE" ]
181    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted⟩

183  GlobalBlock(tid, addr) ≜
184    ∧ AllowGlobal(tid, addr)
185    ∧ LimitAccesses(tid)
```

```
186    ∧ tasks' = [tasks EXCEPT ![addr] = @ ∪ {tid}]
187    ∧ memstate' = [memstate EXCEPT ![addr] =
188        IF  @ = "clean"              THEN "globally_coherent"  ELSE
189        IF  @ = "private_clean"      THEN "ERROR_STATE"        ELSE
190        IF  @ = "private_dirty"      THEN "ERROR_STATE"        ELSE
191        IF  @ = "immutable"          THEN "ERROR_STATE"        ELSE
192        IF  @ = "globally_coherent"  THEN "globally_coherent"  ELSE  "ERROR_STATE" ]
193    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted⟩

195   We need a state transition for the epsilon transition of the global state.
196  ReleaseGlobalBlock(tid, addr) ≜
197    ∧ LimitAccesses(tid)
198    ∧ memstate[addr] = "globally_coherent"
199    ∧ tid ∈ tasks[addr]
200    ∧ tasks' = [tasks EXCEPT ![addr] = @ \ {tid}]
201     If this is the last task dropping the line, we can transition it to clean.
202    ∧ memstate' = IF Cardinality(tasks[addr]) = 1
203                    THEN [memstate EXCEPT ![addr] = "clean"]
204                    ELSE [memstate EXCEPT ![addr] = "globally_coherent"]
205    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted⟩

207 ├──────────────────────────────────────────────────────────────────────┤
208   Cache actions that SW has no control over. Note that these are tied to
209   tids for right now, but there should probably be some concept of a cluster
210   built around them instead.
211  CacheInvalidate(tid, addr) ≜
212    ∧ tid ∈ tasks[addr]
213    ∧ memstate[addr] ∈ {"private_clean", "immutable"}
214    ∧ tasks' = [tasks EXCEPT ![addr] = @ \ {tid}]
215     If there is only one holder of the line, send it back to clean state.
216    ∧ memstate' = IF Cardinality(tasks[addr]) = 1
217                    THEN [memstate EXCEPT ![addr] = "clean"]
218                    ELSE memstate
219    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted, accessesPerTask⟩

221  CacheEviction(tid, addr) ≜
222    ∧ tid ∈ tasks[addr]
223    ∧ memstate[addr] ∈ {"private_dirty"}
224    ∧ tasks' = [tasks EXCEPT ![addr] = @ \ {tid}]
225     If there is only one holder of the line, send it back to clean state.
226    ∧ memstate' = [memstate EXCEPT ![addr] = "clean"]
227    ∧ UNCHANGED ⟨inbarrier, intervalsCompleted, accessesPerTask⟩

229 ├──────────────────────────────────────────────────────────────────────┤
230   ERROR_STATE is the catch all for bad states
231  NoBadState ≜
```

```
232     ∧ ∀ addr ∈ Addr : memstate[addr] ≠ "ERROR_STATE"

234   Ensure that if the block is private, only one task is using it
235  PrivateSafety ≜
236     ∧ ∀ addr ∈ Addr :
237         (memstate[addr] ≠ "private_clean" ∨ memstate[addr] ≠ "private_dirty")
238             ∨ Cardinality(tasks[addr]) = 1

240   Ensure that all data that becomes dirty can eventually become clean
241  Liveness1 ≜
242     ∀ addr ∈ Addr : (memstate[addr] = "private_dirty") ⇒
243       (◇(memstate[addr] = "clean") ∨ □(memstate[addr] = "private_dirty"))

245 |------------------------------------------------------------------------|
246   The next state change may need to change to preculde invalid steps from
247   occuring. One example is forcing local loads to only occur when the current
248   task owns it or it is unowned.
249  Next ≜
250     ∨ (∃ tid ∈ TaskID, addr ∈ Addr :
251         ∨ LLDPrivBlock(tid, addr)
252         ∨ LLDImmuteBlock(tid, addr)
253         ∨ LSTPrivBlock(tid, addr)
254         ∨ InvalidateBlock(tid, addr)
255         ∨ WritebackBlock(tid, addr)
256         ∨ GlobalBlock(tid, addr)
257         ∨ ReleaseGlobalBlock(tid, addr)
258         ∨ CacheInvalidate(tid, addr)
259         ∨ CacheEviction(tid, addr)
260          There are two options here: Eager and Lazy.
261         ∨ CoherenceLazy(tid)
262         )

264  Safety ≜ NoBadState ∧ PrivateSafety
265  Spec ≜ Init
266     ∧ □[Next]_⟨tasks, memstate, inbarrier, accessesPerTask, intervalsCompleted⟩
267  Liveness ≜ Liveness1
268 |------------------------------------------------------------------------|
269  THEOREM Spec ⇒ Liveness
270 └────────────────────────────────────────────────────────────────────────┘
```

REFERENCES

[1] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar, "An integrated quad-core opteron processor," in *IEEE International Solid-State Circuits Conference, 2007. (ISSCC 2007), Digest of Technical Papers*, February 2007, pp. 102–103.

[2] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded SPARC processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[3] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Kottapalli, "A 45nm 8-core enterprise xeon processor," in *IEEE International Solid-State Circuits Conference 2009 (ISSCC 2009), Digest of Technical Papers*, February 2009, pp. 56–57.

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[5] AMD, "The future is fusion: The industry-changing impact of accelerated computing," white paper, 2008.

[6] *Intel CoreTM i7-600, i5-500, i5-400 and i3-300 Mobile Processor Series*, datasheet, Intel, January 2010.

[7] S. Borkar, "Thousand core chips: A technology perspective," in *DAC '07: Proceedings of the 44th Annual Design Automation Conference*, 2007, pp. 746–749.

[8] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume: 1*, Intel Corporation, November 2008.

[9] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming.* Reading, Massachusetts: Addison Wesley, 2008.

[10] Standards Performance Evaluation Corporation, "All published SPEC CPU results," May 2010. [Online]. Available: http://www.spec.org/cpu2006/results/cpu2006.html

[11] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The case for a single-chip multiprocessor,” in *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 2–11.

[12] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel, “Tradeoffs in designing accelerator architectures for visual computing,” in *Proceedings of the International Symposium on Microarchitecture*, 2008, pp. 164–175.

[13] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, “Evolution of thread-level parallelism in desktop applications,” in *ISCA ’10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 302–313.

[14] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, “Larrabee: A many-core x86 architecture for visual computing,” *ACM Transactions on Graphics*, vol. 27, pp. 1–15, 2008.

[15] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y.-K. Chen, “Physical simulation for animation and visual effects: Parallelization and characterization for chip multiprocessors,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 220–231.

[16] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang, “Pangaea: A tightly-coupled ia32 heterogeneous chip multiprocessor,” in *PACT ’08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 52–61.

[17] I. Gelado, J. E. Stone, J. Cabezas, S. J. Patel, N. Navarro, and W.-M. W. Hwu, “An asymmetric distributed shared memory model for heterogeneous parallel systems,” in *ASPLOS’10: Proceedings of the 15th International Conference on Architectural Support for Programming Languagges and Operating Systems*, March 2010, pp. 347–358.

[18] J. H. Kelm and S. S. Lumetta, “HybridOS: Runtime support for reconfigurable accelerators,” in *FPGA’08: Proceedings of the International Symposium on Field-Programmable Gate Arrays*, Februrary 2008, pp. 212–221.

[19] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” in *PPoPP’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 73–82.

[20] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," *SIGARCH Computer Architecture News*, vol. 25, no. 2, pp. 206–218, 1997.

[21] G. F. Pfister, *In search of clusters*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1998.

[22] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research Development*, vol. 46, no. 1, pp. 5–25, 2002.

[23] S. Naffziger, B. Stackhouse, and T. Grutkowski, "The implementation of a 2-core multi-threaded itanium-family processor," in *IEEE International Solid-State Circuits Conference, 2005. (ISSCC2005) Digest of Technical Papers*, vol. 1, February 2005, pp. 182–192.

[24] J. Douglas, "Intel 8xx series and paxville xeon-mp microprocessors," in *Hotchips 17*, August 2005.

[25] M. Golden, S. Arekapudi, G. Dabney, M. Haertel, S. Hale, L. Herlinger, Y. Kim, K. McGrath, V. Palisetti, and M. Singh, "A 2.6ghz dual-core 64bx86 microprocessor with ddr2 memory support," in *IEEE International Solid-State Circuits Conference, 2006. (ISSCC 2006), Digest of Technical Papers*, February 2006, pp. 325–332.

[26] ATI, *ATI Radeon HD 5870 GPU Feature Summary*, 2010. [Online]. Available: http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx

[27] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001, pp. 55–55.

[28] R. J. Rost, *OpenGL(R) Shading Language*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[29] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 2004, pp. 777–786.

[30] *OpenCL Specification*, 1st ed., Khronos OpenCL Working Group, December 2008.

[31] Apple Inc., "Technology brief: Grand central dispatch," August 2009. [Online]. Available: http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf

[32] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 13–23.

[33] J. Gray, "The roadrunner supercomputer: a petaflop's no problem," *Linux Journal*, vol. 2008, no. 175, p. 1, 2008.

[34] M. Toksvig, J. Mathieson, B. Cabral, and B. Smith, "NVIDIA Tegra: Enabling stunning handheld graphics and HD video," in *Hotchips 20*, August 2008.

[35] *Cortex-A9 Technical Reference Manual*, 2nd ed., ARM Ltd., April 2010. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/index.html

[36] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS'67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 1967, pp. 483–485.

[37] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[38] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[39] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 26–36.

[40] S. W. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html

[41] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[42] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the International Symposium on Computer Architecture*, June 2009, pp. 140–151.

[43] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the java hotspot™client compiler for java 6," *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 1, pp. 1–32, 2008.

[44] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, 2004, p. 75.

[45] S. M. Kofsky, D. R. Johnson, J. A. Stratton, W.-M. W. Hwu, S. J. Patel, and S. S. Lumetta, "Implementing a GPU programming model on a non-GPU accelerator architecture," in *A4MMC'10 : 1st Workshop on Applications for Multi and Many Core Processors*, June 2010. [Online]. Available: http://impact.crhc.illinois.edu/ftp/workshop/rcuda.pdf

[46] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 407–420.

[47] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 235–246.

[48] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[49] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[50] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 26–36.

[51] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 235–246.

[52] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak, "The network architecture of the connection machine CM-5," *Journal of Parallel Distributed and Parallel Computing*, vol. 33, no. 2, pp. 145–158, 1996.

[53] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—designing a MIMD, shared-memory parallel machine," in *ISCA'82: Proceedings of the 9th Annual International Symposium on Computer Architecture*, 1982, pp. 239–254.

[54] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "CEDAR: A large scale multiprocessor," *SIGARCH Computer Architecture News*, vol. 11, no. 1, pp. 7–11, 1983.

[55] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, p. 83.

[56] J. Guo, G. Bikshandi, D. Hoeflinger, G. Almasi, B. Fraguela, M. Garzaran, D. Padua, and C. von Praun, "Hierarchically tiled arrays for parallelism and locality," in *IPDPS'06: The 20th International Parallel and Distributed Processing Symposium*, April 2006, pp. 316–323.

[57] M. Gschwind, "Chip multiprocessing and the cell broadband engine," in *Proceedings of the 3rd Conference on Computing Frontiers*, 2006, pp. 1–8.

[58] D. Burger, J. R. Goodman, and A. Kägi, "Memory bandwidth limitations of future microprocessors," in *ISCA '96: Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996, pp. 78–89.

[59] J. Balfour and W. J. Dally, "Design tradeoffs for tiled cmp on-chip networks," in *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, 2006, pp. 187–198.

[60] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[61] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983, pp. 124–131.

[62] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving multiprocessor performance with coarse-grain coherence tracking," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 246–257.

[63] A. Moshovos, "RegionScout: Exploiting coarse grain sharing in snoop-based coherence," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 234–245.

[64] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A scalable architecture based on single-chip multiprocessing," in *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 282–293.

[65] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proceedings of the International Conference on Parallel Processing*, 1990, pp. 312–321.

[66] B. W. O'Krafka and A. R. Newton, "An empirical evaluation of two memory-efficient directory methods," in *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 138–147.

[67] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, vol. 27, no. 12, pp. 1112–1118, 1978.

[68] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 46–56.

[69] M. M. Michael and A. K. Nanda, "Design and performance of directory caches for scalable shared memory multiprocessors," in *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999, p. 142.

[70] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS directories: A scalable cache coherence scheme," in *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 224–234.

[71] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, "Cooperative shared memory: software and hardware for scalable multiprocessors," *ACM Transactions on Computers Systems*, vol. 11, no. 4, 1993.

[72] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 128–138, 2000.

[73] W. W. Carlson and J. M. Draper, "Distributed data access in ac," in *PPoPP '95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995, pp. 39–47.

[74] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 1993, pp. 262–273.

[75] A. Kamil, J. Su, and K. Yelick, "Making sequential consistency practical in Titanium," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 15–26.

[76] W. Carlson, "Introduction to UPC and language specification," IDA Center for Comp. Sci., Tech. Rep. CCS-TR-99-157, 1999.

[77] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. J. Patel, "A task-centric memory model for accelerator architectures," *IEEE Micro*, vol. 30, no. 1, pp. 2–12, January/February 2010.

[78] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 162–173.

[79] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* Sebastapol, California: O'Reilly, 2007.

[80] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[81] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Intel Technology Journal*, February 2005.

[82] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the national Bureau of Standards*, vol. 49, no. 6, pp. 409–436, December 1952.

[83] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. [Online]. Available: http://www.jstor.org/stable/2003354

[84] E. Gilbert, D. Johnson, and S. Keerthi, "A fast procedure for computing the distance between complex objects in three space," in *The Proceedings of the 1987 IEEE International Conference on Robotics and Automation*, March 1987, pp. 1883–1889.

[85] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[86] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 163–169.

[87] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, B. P. Sutton, and Z. P. Liang, "Accelerating advanced mri reconstructions on gpus," *Journal of Parallel Distributed Computing*, vol. 68, no. 10, pp. 1307–1318, 2008.

[88] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. J. Patel, "A task-centric memory model for scalable accelerator architectures," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2009, pp. 77–87.

[89] J. Goodman, "Cache consistency and sequential consistency," IEEE Scalable Interface Working Group, Tech. Rep. 61, March 1989.

[90] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, September 1979.

[91] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *ISCA '86: Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986, pp. 434–442.

[92] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger, "The power of processor consistency," in *SPAA '93: Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 251–260.

[93] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon, "Comparison of hardware and software cache coherence schemes," *SIGARCH Computer Architecture News*, vol. 19, no. 3, pp. 298–308, 1991.

[94] *Software Optimization Guide for AMD family 10h Processors*, 3rd ed., AMD, May 2009.

[95] *Intel 5000X Chipset Memory Controller Hub (MCH)*, 3rd ed., Intel, September 2006.

[96] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 1st ed., NVIDIA, December 2009.

[97] J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, and A. Strong, "A 40nm 16-core 128-thread CMT SPARC SoC processor," in *IEEE International Solid-State Circuits Conference 2010 (ISSCC 2010), Digest of Technical Papers*, February 2010, pp. 98–99.

[98] C. Bienia, S. Kumar, J. P. Singh, and K. Li., "The PARSEC benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-81108, January 2008.

[99] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 280–298.

[100] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA, USA: Morgan-Kaufmann, 2003.

[101] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[102] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, 1996.

[103] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *PPOPP '90: Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 168–176.

[104] S. J. Vaughn-Nichols, "Vendors draw up a new graphics-hardware approach," *IEEE Computer*, vol. 42, no. 5, pp. 11–13, 2009.

[105] *OpenSPARC T2 SoC Microarchitecture Specification Part 1 of 2*, Sun Microsystems Inc., May 2008.

[106] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA highly scalable server," *SIGARCH Computer Architecture News*, vol. 25, no. 2, pp. 241–251, 1997.

[107] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 304–316.

[108] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 261–272.

[109] E. A. Brewer and B. C. Kuszmaul, "How to get good performance from the CM-5 data network," in *Proceedings of the 8th International Symposium on Parallel Processing*, 1994, pp. 858–867.

[110] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 3–14.

[111] R. Friedman, "Implementing hybrid consistency with high-level synchronization operations," in *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of Distributed Computing*, 1993, pp. 229–240.

[112] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 358–368.

[113] OpenMP Architecture Review Board, "OpenMP application program interface," May 2008.

[114] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[115] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A low overhead, software-only approach for fine-grain shared memory," in *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 174–185.

[116] B. Bershad, M. Zekauskas, and W. Sawdon, "The midway distributed shared memory system," *Compcon Spring '93, Digest of Papers*, pp. 528–537, February 1993.

[117] L. Iftode, J. P. Singh, and K. Li, "Scope consistency: A bridge between release consistency and entry consistency," in *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, 1996, pp. 277–287.

[118] B. Choi, R. Komuravelli, H. Sung, R. Bocchino, S. V. Adve, and V. V. Adve, "DeNovo: Rethinking hardware for disciplined parallelism," in *Second USENIX Workshop on Hot Topics in Parallelism (HotPar)*, June 2010.

[119] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," in *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 451–460.

[120] M.-L. Li, R. Sasanka, S. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," in *Proceedings of the IEEE International 2005 Workload Characterization Symposium*, October 2005, pp. 34–45.

[121] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 49–58, 1990.

[122] IEEE, "IEEE standard for scalable coherent interface (SCI)," *IEEE Std. 1596-1992*, August 1993.

[123] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 182–193.

[124] A. Raghavan, C. Blundell, and M. M. K. Martin, "Token tenure: Patching token counting using directory-based cache coherence," in *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 47–58.

[125] A. Gupta and W.-D. Weber, "Cache invalidation patterns in shared-memory multiprocessors," *IEEE Transactions on Computers*, vol. 41, no. 7, pp. 794–810, 1992.

[126] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *SIGARCH Computer Architecture News*, vol. 20, no. 1, pp. 5–44, 1992.

[127] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 544–555.

[128] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364–373.

[129] P. P. Shirvani and E. J. McCluskey, "Padded cache: A new fault-tolerance technique for cache memories," in *VTS '99: Proceedings of the 1999 17th IEEE VLSI Test Symposium*, 1999, pp. 440–451.

[130] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches," in *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006, pp. 155–166.

[131] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004, pp. 288–297.

[132] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 107–116.

[133] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum,

and J. Hennessy, "The stanford flash multiprocessor," in *ISCA '98: 25 years of the International Symposia on Computer Architecture (selected papers)*, 1998, pp. 485–496.

[134] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and typhoon: User-level shared memory," in *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 325–336.

[135] J. DeSouza and L. V. Kalé, "MSA: Multiphase specifically shared arrays," in *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, September 2004.

[136] IBM Staff, *PowerPC Microprocessor 32-bit Family: The Programming Environments*, February 2000.

[137] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, Jun 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1542476.1542525

[138] E. Hagersten and M. Koster, "Wildfire: A scalable path for smps," in *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999, p. 172.

[139] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 184–195.

[140] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *IEEE Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[141] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004, pp. 52–63.

[142] L. Lamport, *Specifying Systems*, 1st ed. Boston, MA: Pearsons Education, 2003.

AUTHOR'S BIOGRAPHY

John Kelm was born in 1983 in Nashua, New Hampshire, and grew up in South Norwalk, Connecticut. He received a bachelor of science and engineering degree in computer science and engineering with a minor in mathematics from the University of Connecticut in May 2005. He received a master of science in electrical and computer engineering from the University of Illinois in December of 2006 and his Doctor of Philosophy in electrical and computer engineering in December 2010. His research interests have covered reconfigurable computing, operating system support for compute accelerators, and parallel architecture. His primary research interests are in hybrid coherence schemes for highly parallel accelerators and chip multiprocessors.

John received the ATI/AMD Fellowship provided by Advanced Micro Devices, and the ECE Distinguished Fellowship. He served as a research assistant for Steven S. Lumetta from 2005 to 2010 and was also assisted in the instruction of undergraduate classes in computer engineering. He was recognized as one of the top teaching assistants by student evaluations. While completing his Ph.D., he interned with Advanced Micro Devices in Sunnyvale, California.