HARDWARE AND SOFTWARE TECHNIQUES FOR SCALABLE THOUSAND-CORE SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Daniel Sanchez Martin

August 2012

This dissertation is online at: http://purl.stanford.edu/mz572jk7876

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William Dally

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Computer architecture is at a critical juncture. Single-thread performance has stopped scaling due to technology limitations and complexity constraints. Manufacturers now rely on multicore processors to scale performance efficiently, and parallel architectures, once rare, are now pervasive across all domains. To keep performance on an exponential curve, the number of cores is expected to increase exponentially, reaching thousands of cores in the next decade. However, achieving efficient thousand-core systems will require significant innovation across the software-hardware stack.

At a high level, two main issues hinder multicore scalability. First, hardware resources must scale efficiently, even as some of them are shared among thousands of threads. In particular, the memory hierarchy is hard to scale in several ways: caches spend considerable energy and latency to implement associative lookups, making them inefficient; conventional cache coherence techniques are prohibitively expensive beyond a few tens of cores; and caches cannot be easily shared among multiple threads or processes. Ideally, software should be able to configure these shared resources to provide good overall performance and quality of service (QoS) guarantees under all possible sharing scenarios. Second, software needs to use these parallel architectures efficiently without burdening the programmer with the complexities of large-scale parallelism. To expose ample parallelism, applications will need to be divided in fine-grain tasks of a few thousand instructions each, and scheduled dynamically in a manner that addresses the three major difficulties of fine-grain parallelism: locality, load imbalance, and excessive overheads.

The focus of this dissertation is to enable efficient, scalable and easy-to-use multicore systems with thousands of cores. To this end, we present contributions that

address both hardware and software scalability bottlenecks. While the overarching goal of these techniques is to enable thousands-core systems, they also improve current systems with tens of cores.

On the hardware side, we present three techniques that, together, enable scalable cache hierarchies that can be shared efficiently. First, ZCache is a cache design that provides high associativity at low cost (e.g., 64-way associativity with the latency, energy and area of a 4-way cache) and is characterized with simple and accurate workload-independent analytical models. We use the high associativity and analytical models of ZCache to develop two techniques that address the scalability problems of shared resources in the cache hierarchy. Vantage implements scalable and efficient fine-grain cache partitioning, which enables hundreds of threads to share caches in a controlled fashion, providing configurability, isolation and QoS guarantees. SCD is a coherence directory that scales to thousands of cores efficiently and causes negligible directory-induced invalidations with minimal overprovisioning, enabling efficient cache coherence with QoS guarantees in large-scale multicores.

On the software side, our contributions enable efficient and scalable dynamic runtimes and schedulers for a wide range of applications and programming models. First, we develop a runtime system that uses high-level information from the programming model about parallelism, locality, and heterogeneity to perform scheduling dynamically and at fine granularity to avoid load imbalance. This runtime can schedule applications with complex dependencies (such as streaming workloads) efficiently and with bounded memory footprint, and outperforms previous schedulers (both static and dynamic) on a wide variety of applications. Unfortunately, dynamic fine-grain runtimes and schedulers are hard to scale beyond tens of threads due to communication and synchronization overheads. We present a combined hardware-software approach to scale these schedulers efficiently. We design ADM, a hardware messaging technique tailored to the needs of scheduling and control applications, and use it to build scalable and efficient hardware-accelerated schedulers that match or outperform hardware-only schedulers and retain the flexibility of software schedulers.

Acknowledgments

First and foremost, I would like to thank my advisor, Christos Kozyrakis, for his mentorship, dedication, and support throughout my graduate studies. Christos has not only been a constant source of excellent, technically sound advice, but also a joy to work with and a true friend. Doing research with him has been immensely rewarding and a great learning experience.

I would also like to thank other faculty for their advice and inspiration. Thanks to Kunle Olukotun for leading the Pervasive Parallelism Lab (PPL), and to other PPL faculty members for their continued guidance and feedback, and for creating such a fruitful research environment: Bill Dally, Mark Horowitz, Pat Hanrahan, Mendel Rosenblum, John Ousterhout, and Subhasish Mitra. I am truly grateful to Mark Hill, my undergraduate advisor at the University of Wisconsin-Madison, for introducing me to computer architecture research and showing me how fun it can be.

I have learned a lot from my fellow graduate students at Stanford. I would like to thank the past and present members of my research group: Jacob Leverich, David Lo, Richard Yoo, Christina Delimitrou, Woongki Baek, Mike Dalton, and Hari Kannan. Jacob has been a great officemate and an amazing sysadmin — without him, my simulations would still be running. It has been a pleasure to collaborate and co-author papers with David and Richard, as well as with George Michelogiannakis and Jeremy Sugerman. Thanks to the PPL students for the many insightful discussions.

I want to thank Bill Dally and Kunle Olukotun for taking the time to be my dissertation readers, Nick Bambos for chairing my defense committee, and our amazing admins, Teresa Lynn and Sue George. I am grateful to Fundacion Caja Madrid and Hewlett-Packard for supporting my graduate studies financially through fellowships.

On a more personal note, I would like to thank my friends, both in the U.S. and in Spain, and my family for their unwavering support and encouragement. My parents, Pedro and Carmen, have been great role models, encouraging me to pursue a Ph.D. at Stanford, and both my parents and brother Dario have stoically put up with me being nine time zones away. Finally, I want to thank Christina for her love and support. Thanks to these people I have been able to finish this dissertation while retaining my sanity, although I know some of them strongly disagree with this statement.

Contents

Chapter 1

Introduction

The microprocessor industry is at a critical juncture. For decades, continuous improvements in device technology have enabled microprocessors to improve single-thread performance exponentially by increasing clock frequency and exploiting instruction-level parallelism. However, both power and design complexity limit further improvements in single-thread performance [5]. As a result, the microprocessor industry has shifted to multicore processors (or chip-multiprocessors, CMPs), which include multiple simpler cores per chip and leverage thread-level parallelism to improve performance efficiently.

The multicore era is now in full swing, and parallel architectures, once exotic and aimed at niche markets, are now pervasive across all domains. All current server, mobile and desktop processors are CMPs [49, 159, 165], and chips with tens of cores are already on the market [79, 141, 153]. To keep system performance on an exponential curve, the number of cores is expected to increase exponentially, reaching a thousand cores in the next decade [22, 102, 136]. However, several issues limit CMP scalability, and addressing them is fundamental to achieve efficient CMPs with thousands of cores and beyond.

1.1 Scalability Challenges

At a high level, two main issues hinder CMP scalability. First, hardware resources must scale efficiently, even as some of them are shared among thousands of threads. Second, software needs to use these parallel architectures efficiently without burdening the programmer with the complexities of large-scale parallelism.

The main hardware scalability bottleneck is the memory hierarchy. CMPs rely on large, multi-level cache hierarchies to mitigate the high cost, high latency and limited bandwidth of main memory. Caches commonly use 50% of chip area and a significant fraction of system energy. Three main problems limit the scalability of these cache hierarchies. First, caches currently spend considerable energy and latency to implement associative lookups, making them inefficient. As we move to more efficient cores and place more pressure on the cache hierarchy, these inefficiencies have a significant impact on system performance. Second, caches are often shared among many threads. This improves utilization, but causes interference and precludes quality of service (QoS) guarantees, and existing techniques to control how sharing is done do not scale beyond a few threads. Third, caches must be kept coherent to make them transparent to software, but traditional coherence schemes do not scale beyond a few tens of cores.

Once we solve all the hardware scalability bottlenecks, we still face the problem of using large-scale CMPs efficiently. To achieve this, software needs to become pervasively parallel without burdening the programmer with the complexities of large-scale parallelism. To expose ample parallelism for hundreds of cores, applications will need to be expressed using fine-grain tasks of a few thousand instructions each, and to be scheduled dynamically in a manner that addresses the three major difficulties of fine-grain parallelism: excessive overheads, load imbalance, and locality. The latter is particularly critical, as communication costs increase with the number of cores and locality optimizations conventionally target coarse-grain tasks.

1.2 Contributions

The focus of this dissertation is to enable efficient, scalable and easy-to-use CMPs with thousands of cores. To this end, our contributions address both hardware and software scalability bottlenecks. On the hardware side, we develop techniques that enable *scalable cache hierarchies that can be shared efficiently*. On the software side, we present contributions that enable *scalable and efficient dynamic fine-grain scheduling* for a large class of applications. While the overarching goal of these contributions is to enable CMPs with thousands of cores, they also improve the performance and efficiency of current CMPs with tens of cores.

Scalable Memory Hierarchies: We present a set of contributions that, together, enable cache hierarchies to scale to thousands of cores efficiently. First, we design a cache that provides high associativity at low cost and is characterized by workload-independent analytical models. We the then use it to implement a scalable cache partitioning technique that enables hundreds of threads to share the cache in a controlled fashion, providing configurability, strict isolation and QoS guarantees, and a coherence directory that scaled to thousands of cores.

Caches implement high associativity to reduce conflict misses. Improving associativity is achieved by increasing the number of ways or positions where each cache line can reside. However, this hurts latency and energy: compared to a 4-way cache, a 32-way cache (used in current CMPs with 6-12 cores) is 30% slower and uses twice the energy per lookup. We have developed *ZCache* [134], a novel cache design that allows much higher associativity than the number of physical ways (e.g., a 64-associative cache with 4 ways). Hits, the common case, *require a single lookup*, incurring the latency and energy of a cache with a small number of ways. On a miss, additional tag lookups are performed off the critical path, yielding an arbitrarily large number of replacement candidates for the incoming line. Using analytical models and extensive simulation, we show that ZCache has two surprising properties. First, *associativity depends only on the number of replacement candidates*, not the number of ways. Second, *simple, workload-independent analytical models* fully characterize the behavior of ZCache. When used as the last-level cache in a 32-core CMP, a 4-way ZCache

improves performance by 7% and full-system energy efficiency by 10% over a 32-way set-associative cache on memory-intensive workloads.

We leverage the analytical guarantees of ZCache to provide efficient controlled sharing in the memory hierarchy. Shared caches introduce interference among the threads sharing the CMP and degrade performance, as infrequently reused data from one thread may displace critical data from another. *Cache partitioning* can be used to solve interference issues, but prior partitioning schemes are limited to a few coarse-grain partitions and reduce associativity, so they do not scale beyond a few cores and degrade performance. We have developed *Vantage* [135], a novel cache partitioning technique that leverages the analytical guarantees of ZCache. Vantage allows *hundreds of fine-grain partitions with capacities defined at line granularity*. Partitions can be dynamically created, deleted and resized efficiently. Vantage is simple to implement, and does not degrade cache performance. It works by partitioning *most* of the cache (e.g., 90%), and controls partition sizes simply by modifying the replacement process. Vantage uses the workload-independent behavior of ZCache and extensive analytical modeling to provide strict guarantees on partition sizes and interference, even though it does not physically partition the cache. Therefore, Vantage enables configurability and QoS guarantees in large-scale CMPs. For example, when partitioning the last-level cache in a 32-core CMP (32 partitions), conventional techniques degrade performance by up to 25% using a barely-implementable 64-way cache, while Vantage improves performance by up to 20% *using a 4-way ZCache*.

Finally, CMPs implement a coherence protocol to provide the illusion of a single level of shared memory, making caches transparent to software. Coherence protocols require a coherence directory to scale beyond a few cores. Directories keep track of cache line sharers, and act as an ordering point for conflicting requests. However, previous directory implementations are not scalable, as they require too much space, excessive bandwidth, or complex changes to the coherence protocol. We leverage ZCache to implement *SCD*, a scalable coherence directory [136]. SCD achieves its scalability by using a variable amount of space in the directory to represent the set of sharers of each line. Lines with one or few sharers use a single tag, and widely shared lines use multiple tags, allowing tags to remain small as the system scales

up. It is area-efficient, fast, simple to implement, and requires no modifications to the coherence protocol. SCD can be characterized with analytical models, and its worst-case energy and performance overheads are bounded, and negligible if sized correctly. Therefore, SCD performs like an ideal directory, providing QoS guarantees regardless of the workload. On a 1024-core CMP, SCD is 13× smaller than a sparse directory; it is also 2× smaller, 20% faster and simpler than a hierarchical directory, the previously known most scalable implementation.

Scalable Dynamic Fine-Grain Scheduling: To achieve load balancing for irregular applications with fine-grain parallelism while maintaining locality and low overheads, we need dynamic schedulers that take advantage of the high-level information about locality and parallelism available in emerging programming models. We have developed such a dynamic scheduling framework for GRAMPS, a programming model that expresses applications as a graph of stages that communicate through queues [137]. GRAMPS has similarities to streaming programming models, but includes mechanisms that allow expressing irregular applications with heterogeneous parallelism. Our runtime leverages programming model information about parallelism, task dependencies and priorities, and producer-consumer communication to improve scheduling. It uses task-stealing with per-stage queues and policies to *maximize locality and schedule fine-grain tasks efficiently, even with complex dependencies*. Moreover, it is the first dynamic scheduler for such programming models that *guarantees bounded memory usage*. It outperforms previous schedulers on a wide range of applications. In a 24-thread system, it is up to 70% faster than a task-stealing dynamic scheduler (used in multi-core programming models such as Cilk, X10, or OpenMP), up to 17× faster than a GPGPU scheduler (used in models such as CUDA or OpenCL), and up to 5.3× faster than static scheduling (used in streaming models such as StreamIt) because it avoids load imbalance.

Nevertheless, communication and synchronization overheads make fine-grain schedulers hard to scale beyond tens of threads. Prior work proposes to address this problem by implementing the scheduler in hardware, but this leads to hard-wired, inflexible scheduling policies. Instead, we propose a hybrid solution that uses hardware

to reduce critical overheads and leaves scheduling policies to software [139]. We have developed *Asynchronous Direct Messages* (ADM), a communication scheme tailored to the needs of scheduling and resource management. ADM enables threads to *send and receive asynchronous, short messages efficiently, bypassing the memory hierarchy.* Low-overhead messaging allows us to implement a family of software-mostly fine-grain schedulers. At 512 threads, they are up to 6.4× faster than software schedulers, and as fast and scalable as a hardware-only scheduler with the same policies. Additionally, the policies of ADM-accelerated schedulers can be tailored to the application, outperforming hardware solutions by up to 70%. ADM is not exposed to the programmer, increasing performance transparently, and is flexible enough to accelerate other primitives (e.g., barriers and interprocess communication).

1.3 Thesis Organization

The rest of this dissertation is organized as follows. Chapter 2 provides relevant background and motivation. Chapter 3 describes ZCache, an efficient highly associative cache that can be characterized using analytical models. In Chapter 4 we present Vantage, a scalable cache partitioning technique. In Chapter 5 we describe SCD, a scalable coherence directory that provides performance guarantees. Chapter 6 presents our dynamic fine-grain scheduler and runtime for GRAMPS, and Chapter 7 presents ADM, a general-purpose hardware mechanism to accelerate scheduling. Finally, Chapter 8 concludes this dissertation.

Chapter 2

Background and Motivation

2.1 Scalable Chip-Multiprocessors

For decades, microprocessor performance has leveraged continuous improvements in CMOS scaling. These improvements can be summarized by Moore's law [115] and the Dennard scaling rules [52]. Moore's law observes that the number of transistors that can be integrated economically on a chip doubles every 24 months. Under classic Dennard scaling, decreasing feature size yields transistors that are smaller, faster, and require lower supply voltage. Therefore, every two years, we could produce chips that had twice the amount of transistors, were clocked 40% faster, and consumed the same amount of power. These chips were area-constrained: the main limitation for performance was the number of transistors and how well they were utilized, not the power consumed.

Architects leveraged CMOS scaling to design increasingly complex uniprocessors that improved single-thread performance exponentially. These processors featured deeper pipelines to increase frequency at an even faster rate than what Dennard scaling provided, and implemented a wide array of techniques to exploit instruction-level parallelism (ILP), such as wide-issue superscalar pipelines, out-of-order execution, and aggressive branch prediction. Many of these features improve performance sub-linearly with respect to area and power. For example, superscalar processors need multi-ported register files, whose cost increases quadratically with the number of

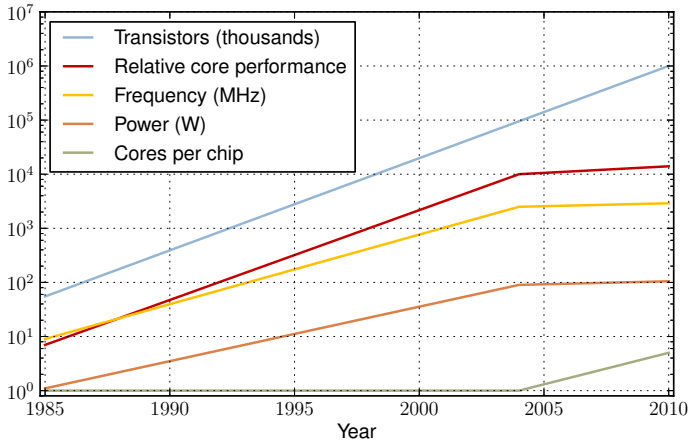


Figure 2.1: Transistors per chip (thousands), relative core performance (normalized SPECint base scores), core frequency (MHz), chip power (W), and cores per chip from 1985 to 2010 [61].

ports, and deep pipelining increases pipeline overheads significantly. Nevertheless, the simultaneous improvements in transistor speed and density provided by Dennard scaling made it possible to deliver microprocessors with twice the performance every two years.

However, multiple trends have progressively made it harder to scale uniprocessors. First, technology scaling is reaching fundamental limits. On one hand, wire delay scales poorly with feature size, so even with faster transistors, building large uniprocessors with tightly-synchronized pipelines becomes harder [5, 21]. On the other hand, supply voltage cannot be scaled anymore because of leakage limitations [55, 150]. Under this scaling regime, we can still produce smaller, faster and more efficient transistors, but doubling the number of transistors at the same frequency requires 40% more power, so designs are now *power-constrained*, not area-constrained. Second, ILP is limited, and all ILP-exploiting techniques provide diminishing returns after a certain point. In 1996, Olukotun et al. [123] already observed that it would be far more efficient to build a chip-multiprocessor (CMP) with several simpler cores, provided the application has enough thread-level parallelism.

In response to these trends, the microprocessor industry has fully shifted to CMPs. As Figure 2.1 shows, single-thread performance is barely scaling anymore, and all

current server, mobile and desktop processors are CMPs [49, 159, 165]. For applications with ample parallelism, chips with a larger number of simple cores are significantly more energy-efficient, so a few CMPs already support hundreds of threads and cores [141, 153]. To keep system performance on an exponential curve, the number of cores is expected to increase exponentially, reaching a thousand cores in the next decade [22, 102, 136]. However, such large-scale CMPs face two main challenges. First, moving to a larger number of simpler core imposes more stringent requirements on the performance and energy efficiency of the memory hierarchy, which is hard to scale beyond a few cores using conventional techniques. Second, software needs to expose massive amounts of parallelism to exploit the large core counts, and maximize locality to use the memory hierarchy efficiently, without burdening the programmer with the complexities of large-scale parallelism. These challenges alone can stop performance scaling of CMPs beyond a few tens of cores, leaving no clear path to future performance increases.

Additionally, in the current power-constrained scaling regime, moving towards simpler cores will eventually not be enough to improve energy efficiency, and the additional efficiency increases required for further performance gains will need other techniques, such as heterogeneous architectures that specialize different resources to specific kinds of computations. In this dissertation, we focus on realizing scalable memory hierarchies and pervasive parallelism on homogeneous CMPs, but these will be needed by heterogeneous architectures just as much (or arguably more, since increasing the efficiency of compute places more stringent demands on the memory hierarchy). Fortunately, the techniques we develop can also apply to heterogeneous architectures. Moreover, scalable memory hierarchies and pervasive parallelism are a more pressing need even under power-constrained scaling, as shown by two recent studies. Esmaeilzadeh et al. [55] consider performance scaling of optimal multicore designs up to 8 nm using conventional PARSEC workloads [16], and conclude that insufficient parallelism, not power constraints, is the main limitation for performance scaling. Hardavellas et al. [72] consider performance scaling assuming almost unbounded parallelism, and find the memory hierarchy to be the main performance bottleneck instead.

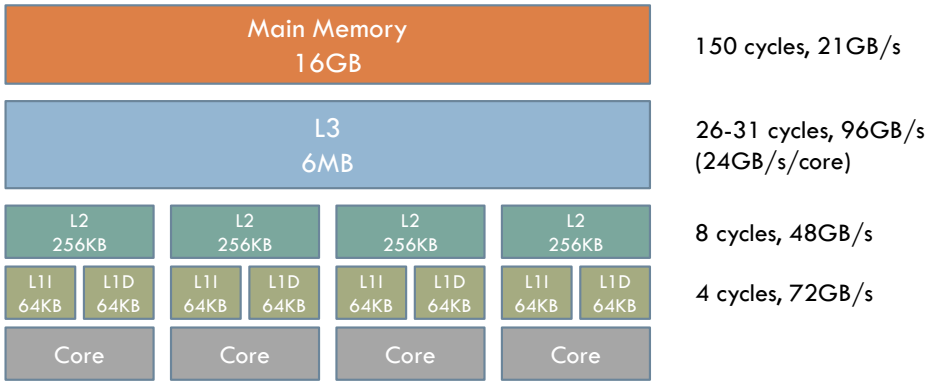


Figure 2.2: Sandy Bridge memory hierarchy, including the size, latency, and bandwidth of each level [168]. Numbers assume an i5-2320 at 3 GHz.

2.2 Memory Hierarchies

Programs often use large amounts of memory, and for performance reasons, memory accesses should be fast. However, cost and physical constraints limit the amount of fast memory. To sidestep with issue, computers rely on a multi-level memory hierarchy, with each level providing a larger amount of slower, cheaper, and denser storage. For example, Figure 2.2 shows the memory hierarchy of a 4-core Nehalem system, with each level providing decreasing bandwidth, and increasing latency and energy per access. Memory hierarchies work because programs exhibit *locality of reference*: although programs use large amounts of memory, most accesses either re-reference data that was accessed recently (temporal locality), or data in nearby locations (spatial locality).

Current general-purpose, small-scale CMPs are commonly built using *caches*. Caches are *associative arrays* that store not only raw data but *(address, data)* pairs. At each level, a memory access first performs a cache lookup. If the data block is not present, it is fetched from the next level of the hierarchy, until it is retrieved, either from an intermediate cache, or from main memory. Caches capture locality implicitly and transparently to software, so they are easier to use than explicitly-addressed, software-managed local stores.

In current small-scale CMPs with tens of cores, the cache hierarchy is organized

as shown in Figure 2.2. First, each core has one or more levels of private caches, which provide fast and energy-efficient access to the critical working set of the running thread. Second, CMPs also include a large, fully shared last-level cache (LLC). A single shared LLC has several advantages over multiple, private LLCs, increasing cache utilization, and providing faster inter-core communication (which happens through the shared cache instead of main memory). Finally, to keep the multiple transfers transparent to software, CMPs implement a coherence protocol that arbitrates communication between the different caches (allowing either multiple read-only copies or a single read-write copy of each cache line).

While the current setup works in CMPs with few cores, several aspects are hard to scale to CMPs with hundreds or thousands of cores. First, large-scale CMPs require caches with high associativity, which is expensive to implement using current techniques. Second, concurrent threads increasingly suffer from interference in the shared LLC, which causes large performance variations, precludes quality-of-service (QoS) guarantees, and degrades shared cache utilization. Current techniques that address interference do not scale beyond few cores. Third, current implementations of cache coherence protocols are hard to scale beyond a few tens of cores, requiring too much area, energy, or complexity. In the rest of this section, we present each of these issues in more detail.

2.2.1 Scalable Cache Associativity

Both caches and coherence directories are commonly built using set-associative arrays. In these arrays, cache associativity, that is, the ability to select a good replacement candidate, is determined by the number of ways. To reduce the number of misses, we would like caches to be highly associative. However, increasing the number of ways also increases cache latency and energy, placing a stringent trade-off on cache design.

Figure 2.3 illustrates the costs of associativity, showing the area, hit latency and hit energy of a set-associative cache as we scale the number of ways from 1 to 32. This cache is optimized for latency $\times$ area $\times$ energy using CACTI 6.5 [116], and consists of a set-associative tag array with 64-bit tags and a data array with 64-byte lines. Lookups

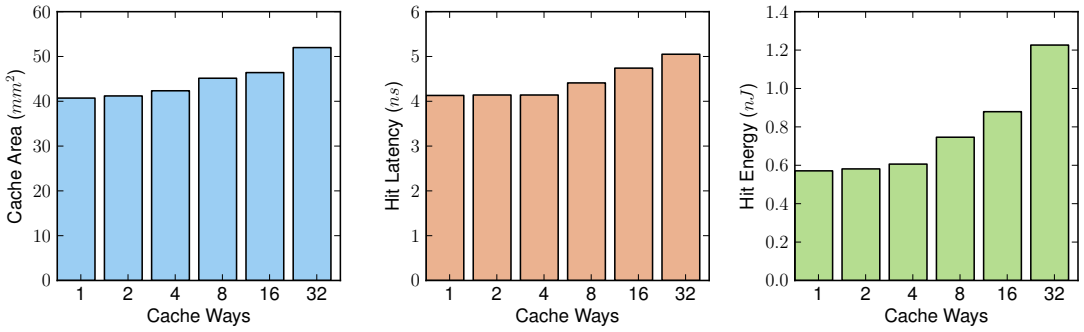


Figure 2.3: Area, hit latency and hit energy of an 8MB set-associative cache array with 1 to 32 ways.

are sequential: first, all the tags in the set are read, and compared with the lookup address; if one of them matches, the line is retrieved from the data array. As we can see, highly associative caches carry a steep cost. Multiple reasons contribute to this: as we increase the number of ways, we need to read and compare more tags in parallel, which requires wider ports, additional logic, and extra latency and energy. While the difference is small with a few ways, it becomes much larger beyond 8 ways. For example, with 32 ways, each hit needs to read 256 bytes of tags — four times the amount of data read (the cache line size).

Unfortunately, as we scale up our systems, two trends make the associativity vs efficiency trade-off much worse. On one hand, the limited bandwidth, high latency, and high energy of memory accesses increasingly demands high associativity to minimize the amount of off-chip accesses. On the other hand, the shift towards a large number of simpler, highly energy-efficient cores means a relatively higher amount of on-chip energy will be consumed in the memory hierarchy, and make cache energy efficiency increasingly important.

Prior research strives to fix the poor efficiency of set-associative arrays with alternative implementations of highly associative arrays. However, most alternative approaches rely on *increasing the number of locations* where a block can be placed (e.g., with multiple locations per way [2, 27, 133], victim caches [14, 86] or extra levels

of indirection [70, 129]). Increasing the number of possible locations of a block ultimately increases the energy and latency of cache hits, and many of these schemes are more complex than conventional cache arrays (requiring e.g., heaps [14], hash-table-like arrays [70] or predictors [27]). Alternatively, good hash can be used to index the cache, spreading out accesses and avoiding worst-case access patterns [93, 140]. While hashing-based schemes improve performance, they are still limited by the number of locations that a block can be in.

Since implementing associative lookups is expensive, we could take a more radical approach and implement the on-chip memory hierarchy explicitly addressed, software-managed *local stores*. Local stores are not associative, so they are are simpler, faster, and more energy-efficient than caches, but must be managed explicitly, either by the programmer or the compiler, This is hard to do efficiently (especially for large memories), makes software composability harder, and requires additional instructions or DMAs to move or copy local store contents. While specialized architectures targeting well-structured programs, such as streaming processors, can work well with a software-managed hierarchy [90, 92, 157], caches are a much better fit for general-purpose CMPs. In fact, in small-scale CMPs, prior research has shown that cache hierarchies achieve similar or better performance and energy efficiency than local stores when using well-known optimizations like prefetching [107].

In Chapter 3, we present ZCache, a cache array that implements high associativity with a small number of physical ways (e.g., achieving 64-way associativity with 4 ways), breaking the trade-off between associativity and access latency or energy.

2.2.2 Scalable Cache Partitioning

While the hardware-managed, software-transparent nature of caches is one of its main advantages, it also comes with a significant drawback: software cannot control how cache space is used, and is at the mercy of the hardware replacement policy. In short, caches are a *best-effort* optimization, providing no QoS guarantees. This problem is intrinsic to all caches, but it is especially insidious in shared caches. While private caches offer predictable and repeatable behavior, when multiple applications

run concurrently on the CMP, they suffer from *interference* in shared caches. This causes large performance variations, precluding QoS guarantees, and can degrade cache utilization, hurting overall throughput. Interference can cause performance variations of over 3× in systems with few cores [80], and is a growing concern due to the increasing number of cores per chip and the emergence of shared compute substrates where this situation is common (e.g., cloud computing).

We can eliminate interference by using *cache partitioning* to divide the cache explicitly among smaller "virtual caches". Partitioning enables software control of on-chip space without reverting to explicitly-managed local stores. Additionally, cache partitioning has several important uses beyond enforcing isolation and QoS in systems with shared caches. For example, in CMPs with private caches, capacity sharing schemes also need to partition each cache [127]. Several software-controlled memory features like virtual local stores [45] or line pinning [117] can be implemented through partitioning. Architectural proposals such as transactional memory and thread-level speculation [32, 71] use caches to store speculative data, and can use partitioning to avoid having that data evicted by non-speculative accesses. Finally, security schemes can use the isolation provided by partitioning to prevent timing side-channel attacks that exploit the shared cache [126].

A cache partitioning solution has two components: an *allocation policy* that decides the size of each partition to achieve a specific objective (e.g., maximize throughput, improve fairness, meet QoS requirements, pin specific data on-chip, etc.), and a *partitioning scheme* that enforces those sizes. While allocation policies are generally simple and efficient [80, 128, 147], current partitioning schemes have serious drawbacks. In this thesis, we focus on the partitioning scheme.

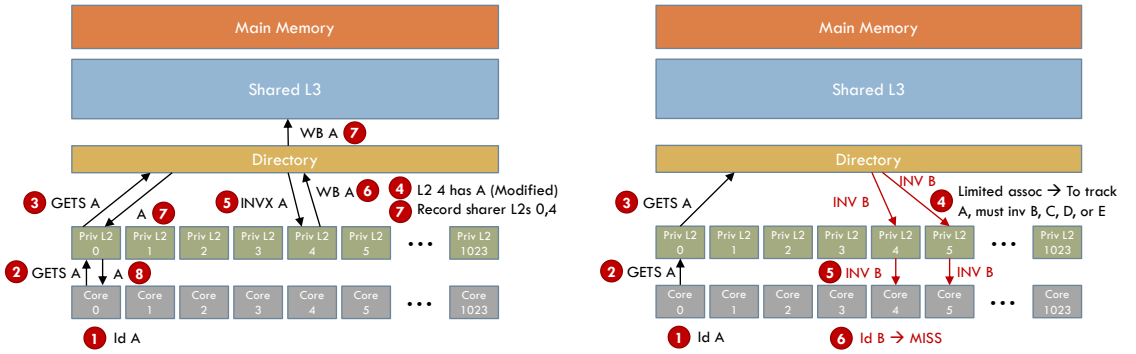
Ideally, a partitioning scheme should satisfy several desirable properties. First, it should be *scalable and fine-grain*, capable of maintaining a large number of fine-grain partitions (e.g., hundreds of partitions of tens or hundreds of lines each). It should maintain *strict isolation* among partitions, with *no reduction of cache performance* (i.e., without hurting associativity or replacement policy performance). It should be *dynamic*, allowing to quickly create, delete or resize partitions. Finally, it should be *simple to implement*.

Unfortunately, prior partitioning schemes fail to meet these properties. Way-partitioning [42] is limited to few coarse-grain partitions (at most, as many partitions as ways) and drastically reduces the associativity of each partition. Other schemes partition the cache by sets instead of ways, either in hardware [132] or software [109], maintaining associativity. However, these methods also lead to coarse-grain partitions, require costly changes to cache arrays and expensive data copying or flushing when partitions are resized, and often do not work with shared address spaces. Finally, proposals such as decay-based replacement [162] or PIPP [163] modify the replacement policy to provide some control over allocations. However, they lack strict control and guarantees over partition sizes and interference, preclude the use of a specific replacement policy within each partition, and are often co-designed to work with a specific allocation policy. Most importantly, current partitioning schemes are *not scalable* or fine-grain. Fully shared LLCs are already used in commercial large-scale CMPs with hundreds of threads and cores [141, 153], and prior research has shown they are desirable in thousand-core CMPs [90], stressing the need for scalable partitioning.

In Chapter 4 we present *Vantage*, a partitioning scheme that overcomes the drawbacks of prior techniques. Vantage can maintain hundreds of partitions defined at cache line granularity, provides strict isolation among partitions, maintains high cache performance, and is simple to implement, working with conventional cache arrays and requiring minimal overheads. Thanks to these features, Vantage enables performance isolation and quality of service in current and future large-scale CMPs, and can be used for several other purposes, such as cache-pinning critical data, or implementing flexible local stores through application-controlled partitions.

2.2.3 Scalable Cache Coherence

Cache coherence is needed to maintain the illusion of a single shared memory on a system with multiple private caches. A coherence protocol arbitrates communication between the private caches and the next level in the memory hierarchy, typically a shared cache (e.g., in a CMP with per-core L2s and a shared last-level cache) or



(a) Load request from core 0 to address A, which is owned by L2 4

(b) Directory-induced invalidations caused by limited directory associativity

Figure 2.4: Examples of (a) a typical directory operation, and (b) directory-induced invalidations.

main memory (e.g., in multi-socket systems with per-die private last-level caches). Implementing coherent cache hierarchies becomes increasingly difficult as the system scales up. Broadcast-based *snooping* coherence protocols work well in small-scale systems, but do not scale beyond a handful of cores due to their large bandwidth overheads, even with optimizations like snoop filters [91]. Large-scale CMPs require a *directory-based* protocol, which introduces a *coherence directory* between the private and shared cache levels to track and control which caches share a line and serve as an ordering point for concurrent requests. Figure 2.4a illustrates this setup, showing how directories introduce an extra level of indirection. Unfortunately, while directory-based protocols scale to hundreds of cores and beyond, implementing directories that can track hundreds of sharers efficiently has been problematic.

Ideally, a directory should satisfy four basic requirements. First, it should maintain sharer information while imposing small area, energy and latency overheads that scale well with the number of cores. Second, it should be simple to implement, requiring no changes to the coherence protocol. Third, it should represent sharer information accurately — it is possible to improve directory efficiency by allowing inexact sharer information, but this causes additional traffic and complicates the coherence protocol. Fourth, it should introduce a negligible amount of directory-induced

invalidations (those due to limited directory capacity or associativity, as shown in Figure 2.4b), as they can significantly degrade performance.

Proposed directory organizations make different trade-offs in meeting these properties, but no scheme satisfies all of them. Traditional schemes scale poorly with core count: Duplicate-tag directories [13, 149] maintain a copy of all tags in the tracked caches. They incur reasonable area overheads and do not produce directory-induced invalidations, but their highly-associative lookups make them very energy-inefficient with a large number of cores. Sparse directories [69] are associative, address-indexed arrays, where each entry encodes the set of sharers, typically using a bit-vector. However, sharer bit-vectors grow linearly with the number of cores, making them area-inefficient in large systems, and their limited size and associativity can produce significant directory-induced invalidations. For this reason, set-associative directories tend to be significantly oversized [59]. There are two main alternatives to improve sparse directory scalability. Hierarchical directories [158, 164] implement multiple levels of sparse directories, with each level tracking the lower-level sharers. This way, area and energy grow logarithmically with the number of cores. However, hierarchical organizations impose additional lookups on the critical path, hurting latency, and more importantly, require a more complex hierarchical coherence protocol [158]. Alternatively, many techniques have been explored to represent sharer sets inexactly through coarse-grain bit-vectors [69], limited pointers [3, 33], Tagless directories [170] and SPACE [172]. Unfortunately, these methods introduce additional traffic in the form of spurious invalidations and often increase coherence protocol complexity [170].

Since cache coherence is hard to scale using conventional techniques, we could simply not implement it. However, this complicates software, which must either manage the private caches explicitly (e.g., flushing lines or the whole cache to make updates globally visible) or explicitly access the shared cache (bypassing the private caches) to implement communication and synchronization. While this model may work for special-purpose accelerators with restricted programming models and applications with highly structured communication and synchronization [], prior work shows that hardware cache coherence is required in practice in general-purpose CMPs [91, 102].

Hundred-core CMPs are already on the market, stressing the need for scalable coherence.

In Chapter 5 we present the Scalable Coherence Directory (SCD), a novel directory scheme that scales to thousands of cores efficiently, while incurring negligible invalidations and keeping an exact sharer representation.

2.2.4 Analytical Design

Finally, the conventional design approach in computer architecture has several drawbacks for large-scale CMPs. The primary goal of computer architects is to improve performance by focusing on common-case behavior. Architects implement a wide array of empirical *best-effort* techniques (such as caching and branch prediction) and provision them by looking at patterns from past and current workloads. To provision for future workloads, or to mitigate performance cliffs and worst-case behavior, resources are often overprovisioned (e.g., implementing a cache with slightly more capacity or associativity).

While this approach worked well for uniprocessors, multicores have a fundamental problem with it: because many resources are shared, we need to provide guarantees on all sharing scenarios if we want to provide a certain level of performance for a specific application. Overprovisioning alone is insufficient and wasteful to guarantee good performance: although some overprovisioning simplifies system design, we need techniques that provide guarantees with minimal overprovisioning.

The root cause of this issue is that computer architecture design is mostly done empirically: system components are assumed to be too complex to be described with useful analytical models, and design mainly relies on simulation to show whether a technique works. Analytical models, if ever, are derived a posteriori.

A major result of this thesis is to show that, against conventional wisdom, an analytical design approach is both practical and highly beneficial. Instead of assuming that components are too complex to be modeled correctly, we design them so that they can be described using simple and accurate analytical models, and then use these models to build techniques that *guarantee* scalability and QoS under all

scenarios while also outperforming conventional techniques in the common case. In particular, we design ZCache so that its associativity is characterized by simple, workload-independent models, and leverage this property in Vantage and SCD to provide QoS guarantees. For example, Vantage gives strong guarantees on partition sizes, isolation and associativity regardless of partition behavior, even though it does not physically partition the cache, and SCD provides guarantees on performance and negligible invalidations regardless of the workload with minimal overprovisioning. While Vantage and SCD can be implemented using set-associative arrays, using ZCache enables them to provide performance guarantees.

2.3 Parallel Runtimes

Even with scalable parallel architectures, we are still faced with the problem of utilizing them efficiently. This requires making software pervasively parallel without imposing too much complexity on the programmer. To this end, applications should be written in high-level parallel programming models such as Cilk [60], TBB [81], CUDA [122], OpenCL [94], StreamIt [151], and Delite [26]. These models provide constructs to express parallelism and synchronization in a manageable way, and take care of resource management and scheduling for the programmer.

While high-level programming models simplify parallel programming, they require a parallel runtime that implements scheduling and resource management. In this thesis, we focus on developing techniques to scale these runtimes efficiently.

2.3.1 Efficient Parallel Runtimes

A runtime should satisfy four desirable properties. First, it should keep the execution units well utilized, performing load balancing if needed. Second, it should keep the memory hierarchy well utilized, producing schedules that exploit application locality and minimize remote or off-chip memory accesses. Third, it should guarantee bounds on the resources consumed. In particular, bounding memory footprint is especially important, as this enables allocating off-chip and on-chip memory resources, avoids

thrashing and out-of-memory conditions, and reduces cache misses (in cache-based systems) or spills to main memory (in systems with explicitly managed local stores). Fourth, it should impose small scheduling overheads.

The ability of the scheduler to realize these properties is constrained by the information available from the programming model (e.g., locality hints, task dependencies, types of parallelism exploited). Consequently, schedulers are often tailored to a specific programming model. Unfortunately, schedulers that satisfy these properties are only available for limited programming models.

Most scheduling approaches can be broadly grouped into three categories. First, *Task-Stealing* is popular in general-purpose multi-cores, and is used in Cilk, TBB, X10 [34], OpenMP [53], among others. It imposes small overheads, and some programming models, such as Cilk and X10, can bound memory footprint by tailoring its scheduling policies [4, 19]. However, it does not leverage program structure and does not work well when tasks have complex dependencies, so it has difficulties scheduling complex pipeline-parallel applications. Second, *Breadth-First* is used in GPGPU models such as CUDA and OpenCL, and focuses on extracting data parallelism, but cannot exploit task and pipeline parallelism and does not bound memory footprint. Third, *Static* is common in streaming architectures and stream programming models like StreamIt and StreamC/KernelC [50]. It relies on a priori knowledge of the application graph to statically generate an optimized schedule that uses bounded memory footprint [104]. Unfortunately, Static schedulers forgo run-time load balancing, and work poorly when the application is irregular or the architecture has dynamic variability, thus limiting their utility.

As we can see, none of these scheduling techniques satisfies the desirable properties for a wide range of programming models. Programming models with simple or no dependencies can use dynamic schedulers (Task-Stealing or Breadth-First), but as soon as the model has richer semantics and allows for involved dependencies (e.g., pipelines or ordered streams), we are constrained to a static scheduler.

In Chapter 6 we develop a series of techniques to solve this problem, enabling schedulers that satisfy all the desirable properties for richer programming models with complex dependencies. We demonstrate these techniques by implementing a runtime

for GRAMPS [146], a programming model designed to support dynamic scheduling of pipeline and data parallelism. Similar to streaming models, GRAMPS applications are expressed as a graph of stages that communicate either explicitly through data queues or implicitly through memory buffers. However, GRAMPS introduces several enhancements that allow dynamic scheduling and applications with irregular parallelism. Compared to Task-Stealing models, knowing the application graph gives two main benefits. First, the graph contains all the producer-consumer relationships, enabling locality optimizations. Second, memory footprint is easily bounded by limiting the size of queues and memory buffers. However, prior work [146] was based on an idealized simulator with no scheduling overheads, making it an open question whether a practical GRAMPS runtime could be designed. We show that our GRAMPS scheduler achieves significant benefits over existing scheduling techniques on applications from a variety of domains, enabling dynamic scheduling of pipeline-parallel programs with non-trivial dependencies. While our implementation and evaluation focuses on GRAMPS, the techniques developed are directly applicable to other programming models with streaming characteristics, like StreamIt [151] and Delite [26].

2.3.2 Scalable Fine-Grain Scheduling

As we will see in Chapter 6, efficient dynamic schedulers have significant advantages over static schedulers. However, as the number of cores scales up, these schedulers need finer-grain tasks to expose enough parallelism, and scheduling overheads often become a significant bottleneck. We can address this by implementing the scheduler in hardware. GPUs already implement hardware schedulers [130], and Carbon [101] proposes to implement task-stealing in hardware using specialized queues and a custom messaging protocol for enqueuing, dequeuing and distributing tasks. While this solves the performance bottleneck, it introduces two significant problems. First, hardware schedulers fix the scheduling algorithm at design time, making it difficult to accelerate an application or programming model that requires a different algorithm. While supporting some variations is feasible, implementing all the possible algorithms and options in hardware is prohibitively expensive in terms of design and verification

complexity. Second, hardware schedulers introduce a significant amount of custom hardware that cannot be used for other purposes. Ideally, we would like to accelerate scheduling using general primitives that can be leveraged by different algorithms and have other uses beyond scheduling.

In Chapter 7 we present a combined hardware-software approach to build fine-grain schedulers that retain the flexibility of software schedulers while being as fast and scalable as hardware ones. We find that software schedulers are limited by communication and synchronization overheads, not computation. Therefore, we introduce communication support tailored to the needs of scheduling and control applications. We propose asynchronous direct messages (ADM), a simple architectural extension that provides direct exchange of asynchronous, short messages between threads in the CMP without going through the memory hierarchy. ADM is sufficient to implement a family of novel, software-mostly schedulers that rely on low-overhead messaging to efficiently coordinate scheduling and transfer task information. These schedulers match and often exceed the performance and scalability of hardware schedulers when using the same scheduling algorithm, but can tailor their scheduling algorithm to application characteristics or different programming models.

Chapter 3

ZCache: Decoupling Ways and Associativity

3.1 Introduction

As we discussed in Section 2.2, conventional cache implementations suffer from two significant problems. First, *highly-associative caches are inefficient*, as previously proposed implementations rely on increasing the number of ways or locations where a line can be placed. This reduces conflict misses, but increases hit latency and energy, placing a stringent trade-off on cache design. Second, *cache associativity is highly workload-dependent*, and designers have no analytical tools to reason about associativity, so caches are provisioned empirically, by extensive simulation, and provide no associativity guarantees.

In this chapter we present zcache, a cache design with two crucial properties. First, zcaches achieve *arbitrarily high associativity with a small number of physical ways*, breaking the trade-off between associativity and access latency or energy. Second, zcache's *associativity is fully characterized using simple, accurate and workload-independent analytical models*, enabling designers to provision the cache analytically and providing the foundation for techniques that implement QoS guarantees efficiently through analytical design, such as Vantage (Chapter 4) and SCD (Chapter 5). This chapter presents three main contributions:

1. We describe the zcache design. ZCache improves associativity while keeping the number of possible locations of each block small. ZCache's design is motivated by the insight that *associativity is the ability of a cache to select a good block to evict on a replacement*. For instance, given an access pattern with high temporal locality, the best block to evict is the least recently used one in the entire cache. A cache that provides a high-quality stream of evicted blocks essentially has higher associativity, regardless of the number of locations each block can be placed in. Like a skew-associative cache [140], a zcache accesses each way using a different hash function. A block can be in only one location per way, so *hits, the common case, require only a single lookup*. On a replacement, ZCache exploits that with different hash functions, a block that conflicts with the incoming block can be *moved* to a non-conflicting location in another way instead of being evicted to accommodate the new block. This is similar to cuckoo hashing [124], a technique to build space-efficient hash tables. On a miss, the zcache walks the tag array to obtain additional replacement candidates, evicts the best one, and performs a few moves to accommodate the incoming block. This happens off the critical path, concurrently with the miss and other lookups, so it has no effect on access latency.
2. We develop an *analytical framework* to understand associativity and compare the associativities of different cache designs independently of the replacement policy. We define associativity as a probability distribution and show that, under a set of conditions, which are met by zcaches, associativity depends only on the number of replacement candidates. Therefore, we prove that zcache *decouples associativity from the number of ways* (or locations that a block can be in).
3. We evaluate using zcaches at the last-level cache of the CMP's memory hierarchy. Using our analytical framework we show that, for the same number of ways, zcaches provide higher associativity than set-associative caches for most workloads. We also simulate a variety of multithreaded and multiprogrammed workloads on a large-scale CMP, and show that zcaches achieve the benefits of highly-associative caches without increasing access latency or energy. For example, over a set of 10 miss-intensive workloads, a 4-way zcache provides 7% higher IPC and 10% better energy efficiency than a 32-way set-associative cache.

3.2 Background on Cache Associativity

Apart from simply increasing the number of ways in a cache and checking them in parallel, there is abundant prior work on alternative schemes to improve associativity. They mainly rely on either using hash functions to spread out cache accesses, or increasing the number of locations that a block can be in.

3.2.1 Hashing-based Approaches

Hash block address: Instead of using a subset of the block address bits as the cache index, we can use a better hash function on the address to compute the index. Hashing spreads out access patterns that are otherwise pathological, such as strided accesses that always map to the same set. Hashing slightly increases access latency as well as area and power overheads due to this additional circuitry. It also adds tag store overheads, since the full block address needs to be stored in the tag. Simple hash functions have been shown to perform well [93], and commercial processors often implement this technique in their last-level cache [149].

Skew-associative caches: Skew-associative caches [140] index each way with a different hash function. A specific block address conflicts with a fixed set of blocks, but those blocks conflict with other addresses on other ways, further spreading out conflicts. Skew-associative caches typically exhibit lower conflict misses and higher utilization than a set-associative cache with the same number of ways [20]. However, they have no sets, so they cannot use replacement policy implementations that rely on set ordering (e.g., using pseudo-LRU to approximate LRU).

3.2.2 Approaches that Increase the Number of Locations

Allow multiple locations per way: Column-associative caches [2] extend direct-mapped caches to allow a block to reside in two locations based on two (primary and secondary) hash functions. Lookups check the second location if the first is a miss and a rehash bit indicates that a block in the set is in its secondary location. To improve access latency, a hit in a secondary location causes the primary and

secondary locations to be swapped. This scheme has been extended with better ways to predict which location to probe first [27], higher associativities [171], and schemes that explicitly identify the less used sets and use them to store the more used ones [133]. The drawbacks of allowing multiple locations per way are the variable hit latency and reduced cache bandwidth due to multiple lookups, and the additional energy required to do swaps on hits.

Use a victim cache: A victim cache is a highly or fully-associative small cache that stores blocks evicted from the main cache until they are either evicted or re-referenced [86]. It avoids conflict misses that are re-referenced after a short period, but works poorly with a sizable amount of conflict misses in several hot ways [25]. Scavenger [14] divides cache space into two equally large parts, a conventional set-associative cache and a fully-associative victim cache organized as a heap. Victim cache designs work well as long as misses in the main cache are rare. On a miss in the main cache, they introduce additional latency and energy consumption to check the victim cache, regardless of whether the victim cache holds the block.

Use indirection in the tag array: An alternative strategy is to implement tag and data arrays separately, making the tag array highly associative, and having it contain pointers to a non-associative data array. The Indirect Index Cache (IIC) [70] implements the tag array as a hash table using open-chained hashing for high associativity. The V-Way cache [129] implements a conventional set-associative tag array, but makes it larger than the data array to make conflict misses rare. Tag indirection schemes suffer from extra hit latency, as they are forced to serialize accesses to the tag and data arrays. Both the IIC and the V-Way cache have tag array overheads of around 2×, and the IIC has a variable hit latency.

Several of these designs both increase cache associativity and propose a new replacement policy, sometimes tailored to the proposed design [14, 70, 129, 140]. This makes it difficult to elucidate how much improvement is due to the higher associativity and how much depends on the better replacement policy. Instead, we consider that associativity and replacement policy are separate issues, and focus on associativity.

3.3 ZCache Design

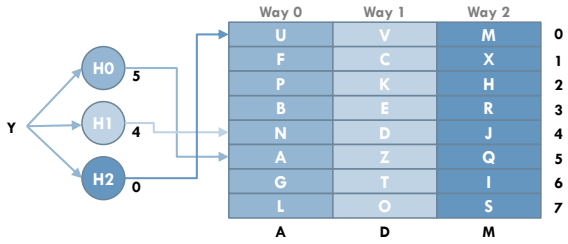
Structurally, zcaches share many common elements with skew-associative caches. Each way is indexed by a different hash function, and a cache block can only reside in a single position on each way. That position is given by the hash value of the block's address. Hits happen exactly as in skew-associative caches, requiring a single lookup to a small number of ways. On a miss, however, zcache exploits the fact that two blocks that conflict on a way often do not conflict on the other ways to increase the number of replacement candidates. ZCache performs a replacement over multiple steps. First, it *walks* the tag array to identify the set of replacement candidates. It then picks the candidate preferred by the replacement policy (e.g., least recently used block for LRU), and evicts it. Finally, it performs a series of *relocations* to be able to accommodate the incoming block at the right location.

The multi-step replacement process happens while fetching the incoming block from the memory hierarchy, and does not affect the time required to serve the miss. In non-blocking caches, simultaneous lookups happen concurrently with this process. The downside is that the replacement process requires extra bandwidth, especially on the tag array, and needs extra energy. However, should bandwidth or energy become an issue, the replacement process can be stopped early, simply resulting in a worse replacement candidate.

3.3.1 Operation

We explain the operation of the replacement process in detail using the example in Figure 3.1. The example uses a small 3-way cache with 8 lines per way. Letters A-Z denote *cache blocks*, and numbers denote *hash values*. Figure 3.1g shows the timeline of accesses to the tag and data arrays, and the memory bus. Throughout Figure 3.1, addresses and hash values obtained in the same access are shown in the same color.

Walk: Figure 3.1a shows the initial contents of the cache and the miss for address Y that triggers the replacement process. Initially, the addresses returned by the tag lookup for Y are our only replacement candidates for the incoming block (addresses



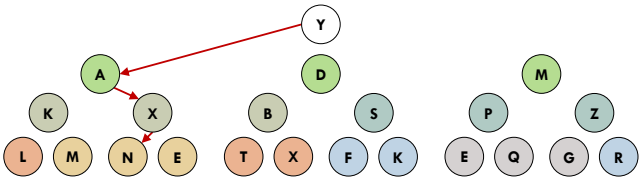
(a) Initial state of the cache and initial miss

Addr	Y	A	D	M
H0	5	5	3	2
H1	4	2	4	5
H2	0	1	7	0

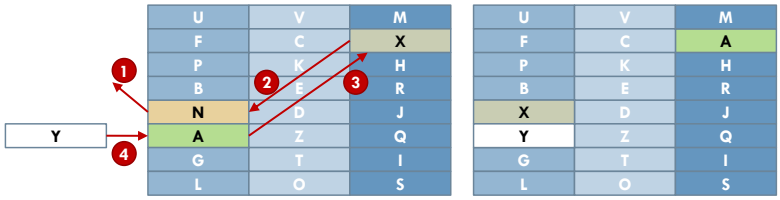
(b) First-level candidates

B	K	X	P	Z	S	Addr
3	7	4	2	6	1	H0
6	2	3	3	5	2	H1
1	0	1	5	3	7	H2

(c) Second-level candidates

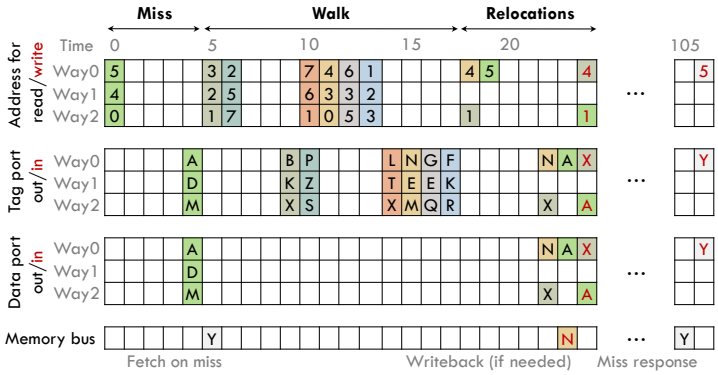


(d) The three levels of replacement candidates. N is selected by the replacement policy



(e) Relocations done to accommodate the incoming block

(f) Final cache state after replacement



(g) Timeline of requests and responses

Figure 3.1: ZCache replacement process

A, D and M). These are the *first-level* candidates. A skew-associative cache would only consider these candidates. In a zcache, the controller starts the walk to expand the number of candidates by computing the hash values of these addresses, shown in Figure 3.1b. One of the hash values always matches the hash value of the incoming block. The others denote the positions in the array where we could *move* each of our current replacement candidates to accommodate the incoming block. For example, as column A in Figure 3.1b shows, we could move block A to line 2 in way 1 (evicting K) or line 1 in way 2 (evicting X) and write incoming block Y in line 5 of way 0.

We take the six non-matching hash values in Figure 3.1b and perform two accesses, giving us an additional set of six *second-level* replacement candidates, as shown in Figure 3.1c (addresses B, K, X, P, Z, and S). We can repeat this process (which, at its core, is a breadth-first graph walk) indefinitely, getting more and more replacement candidates. In practice, we eventually need to stop the walk and select the best candidate found so far. In this example, we expand up to a third level, reaching 21 (3+6+12) replacement candidates. In general, it is not necessary to obtain full levels. Figure 3.1d shows a tree with the three levels of candidates. Note how, in expanding the second level, some hash values are repeated and lead to the same address. These *repeats* are bound to happen in this small example, but are very rare in larger caches with hundreds to thousands of blocks per way. For example, on a 3MB, 3-way, 21-candidate zcache as in the example, using 64-byte lines (16384 lines/way), only 0.4% of walks have one or more repeats.

Relocations: Once the walk finishes, the replacement policy chooses the best replacement candidate. We discuss the implementation of replacement policies in Section 3.3.5. In our example, block N is the best candidate, as shown in Figure 3.1d. To accommodate the incoming block Y, the zcache evicts N and relocates its ancestors in the tree (both data and tags), as shown in Figure 3.1e. This involves reading and writing the tags and data to their new locations, as the timeline in Figure 3.1g indicates. Figure 3.1f shows the cache contents after the replacement process is finished, with N evicted and Y in the cache. Note how N and Y both used way 0, but completely different locations.

3.3.2 General Figures of Merit

A zcache with W ways where the walk is limited to L levels has the following figures of merit:

- Replacement candidates (R): Assuming no repeats when expanding the tree, $R = W \sum_{l=0}^{L-1}(W-1)^l$.
- Replacement process energy (E_{miss}): If the energies to read/write tag or data in a single way are denoted E_{rt}, E_{wt}, E_{rd} and E_{wd}, then $E_{miss} = E_{walk} + E_{relocs} = R \times E_{rt} + m \times (E_{rt} + E_{rd} + E_{wt} + E_{wd})$, where $m \in \{0, .., L-1\}$ is the number of relocations. Note that reads and writes to the data array, which consume most of the energy, grow with L, i.e., *logarithmically* with R.
- Replacement process latency: Because accesses in a walk can be pipelined, the latency of a walk grows with the number of levels, unless there are so many accesses on each level that they fully cover the latency of a tag array read: $T_{walk} = \sum_{l=0}^{L-1} \max(T_{tag}, (W-1)^l)$. This means that, for $W > 2$, we can get tens of candidates in a small amount of delay. For example, Figure 3.1g assumes a tag read delay of 4 cycles, and shows how the walk process for 21 candidates (3 levels) completes in $4 \times 3 = 12$ cycles. The whole process finishes in 20 cycles, much earlier than the 100 cycles used to retrieve the incoming block from main memory.

3.3.3 Implementation

To implement the replacement process, the cache controller needs some modifications involving hash functions, some additional state and, for non-blocking caches, scheduling of concurrent operations.

Hash functions: We need one hash function per way. Hash functions range from extremely simple (e.g., bit selection) to exceedingly complex (e.g., cryptographic hash functions like SHA-1). In this study, we use H_3 hash functions [31], a family of low-cost, universal, pairwise-independent hash functions that require a few XOR gates per hash bit [138]. We choose H_3 functions because of their good analytical properties and simple implementation, although other hash function families may also work well with zcache.

State: The controller needs to remember the positions of the replacement candidates visited during the walk and the position of the best eviction candidate. Tracking only the most desirable replacement candidate is not sufficient, because relocations need to know about all blocks in the path. However, a single-ported SRAM or small register file suffices. This memory does not need to store full tags, just R hash values. Also, no back-pointers need to be stored, because for a certain position in the SRAM, the parent's position is always the same. In the example shown in Figure 3.1, the controller needs 63 bits of state to track candidates (21 hash values $\times$ 3 bits/value). If the cache was larger, e.g., 3MB, with 1MB per way and 64-byte lines (requiring 14 bits/hash value), it would need 294 bits. Additionally, the controller must buffer the tags and data of the L lines it reads and writes on a relocation. Since the number of levels is typically small (2 or 3 in our experiments), this also entails a small overhead.

Concurrent operations for non-blocking caches: To avoid increasing cache latency, the replacement process should be able to run concurrently with all other operations (tag/data reads and writes due to hits, write-backs, invalidations, etc.). The walk process can run concurrently without interference. This may lead to benign races where, for example, the walk identifies the best eviction candidate to be a block that was accessed (e.g., with a hit) in the interim. This is exceedingly rare in large caches, so we simply evict the block anyway. In smaller caches (e.g., highly-associative but small TLBs or first-level caches), we could keep track of the best two or three eviction candidates and discard them if they are accessed while the walk process is running.

In the second part of the replacement, the relocations, the controller must block intervening operations to at most L positions while blocks in these positions are being relocated. We note that the controller already has logic to deal with these cases (e.g., with MSHRs [97]).

While it is feasible to run multiple replacement processes concurrently, it would complicate the cache controller, and since replacements are not in the critical path, they can simply queue up. Even in caches that serve many concurrent misses, MSHRs can be used to satisfy each miss before finishing its corresponding replacement and

filling in the data. Concurrent replacements would only make sense to increase bandwidth utilization when the cache is close to bandwidth saturation. As we will see in Section 3.6, we do not see the need for such mechanism in our experiments.

In conclusion, zcache imposes minor state and logic overheads to traditional cache controllers.

3.3.4 Extensions

We now discuss additional implementation options to enhance zcaches.

Avoiding repeats: In small first-level caches or TLBs, repeats can be common due to walking a significant portion of the cache. Moreover, a repeat at a low level can trigger the expansion of many repeated candidates. Repeats can be avoided by inserting the addresses visited during the walk in a Bloom filter [18], and not continuing the walk through addresses that are already represented in the filter. Repeats are rare in our experiments, so we do not see any performance benefit from this.

Alternative walk strategies: The current walk performs a breadth-first search for candidates, fully expanding all levels. Alternatively, we could perform a depth-first search (DFS), always moving towards higher levels of replacement candidates. Cuckoo hashing [124] follows this strategy. DFS allows us to remove the walk table and interleave walk with relocations, reducing state. However, it increases the number of relocations for a given number of replacement candidates (since $L = R/W$), which in turn increases both the energy required per replacement (as relocations read and write to the much wider data array) and replacement latency (as accesses in the walk cannot be pipelined). BFS is a much better match to a hardware implementation as the extra required state for BFS is a few hundred bits at most. Nevertheless, a controller can implement a hybrid BFS+DFS strategy to increase associativity cheaply. For instance, in our example in Figure 3.1, the controller could perform a second phase of BFS, trying to re-insert N rather than evicting it, to double the number of candidates without increasing the state needed.

3.3.5 Replacement Policy

So far, we have purposely ignored how the replacement policy is implemented. In this section, we cover how to implement or approximate LRU. While set-associative caches can cheaply maintain an order of the blocks in each set (e.g., using LRU or pseudo-LRU), since zcaches do not have sets, policies that rely on set ordering need to be implemented differently. However, several processor designs already find it too expensive to implement set ordering and resort to policies that do not require it [74, 149]. Additionally, some of the latest, highest-performing policies do not rely on set ordering [85]. While designing a replacement policy specifically tailored to zcaches is an interesting endeavor, we defer it to future work.

Full LRU: We use a global timestamp counter, and add a timestamp field to each block in the cache. On each access, the timestamp counter is incremented, and the timestamp field is updated to the current counter value. On a replacement, the controller selects the replacement candidate with the lowest timestamp (in $\mod 2^n$ arithmetic). This design requires very simple logic, but timestamps have to be large (e.g., 32 bits) to make wrap-arounds rare, thus having high area overhead.

Bucketed LRU: To decrease space overheads, we can make timestamps smaller, and have the controller increase the timestamp counter once every k accesses. For example, with $k = 5\%$ of the cache size and $n = 8$ bits per timestamp, it is rare for a block to survive a wrap-around without being either accessed or evicted. We use this LRU policy in our evaluation.

3.4 Analytical Framework for Associativity

Quantifying and comparing associativity across different cache designs is hard. In set-associative caches, more ways implicitly mean higher associativity. However, when comparing different designs (e.g., a set-associative cache and a zcache), the number of ways becomes a useless proxy for associativity.

The most commonly used approach to quantify associativity is by the number

of conflict misses [75]. Conflict misses for a cache are calculated by subtracting the number of misses incurred by a fully-associative cache of the same size from the total number of misses. Using conflict misses as a proxy for associativity has the advantage of being an end-to-end metric, directly linking associativity to performance. However, it is subject to three problems. First, it is highly dependent on the replacement policy; for example, by using an LRU replacement policy in a workload with an anti-LRU access pattern, we can get higher conflict misses when increasing the number of ways. Second, in CMPs with multilevel memory hierarchies, changing the associativity can alter the reference stream at higher cache levels, and comparing the number of conflict misses when the total number of accesses differs is meaningless. Finally, conflict misses are workload-dependent, so they cannot be used as a general proxy for associativity.

In this section, we develop a framework to address these issues, with the objectives of (1) being able to compare associativity between different cache organizations, and (2) determining how various design aspects (e.g., ways, number of replacement candidates, etc.) influence cache associativity.

3.4.1 Associativity Distribution

Model: We divide a cache into the following components:

- Cache array: Holds tags and data, implements associative lookups by block address, and, on a replacement, gives a list of replacement candidates that can be evicted.
- Replacement policy: Maintains a *global* rank of which cache blocks to replace.

This model assumes very little about the underlying cache implementation. The array could be set-associative, a zcache, or any of the schemes mentioned in Section 3.2. The only requirement that we impose on the replacement policy is to define a global ordering of blocks, which most policies conceptually do. For example, in LRU blocks are ranked by the time of their last reference, in LFU they are ordered by access frequency, and in OPT [15] they are ranked by the time to their next reference. This does not mean that the implementation actually maintains this global rank. In a set-associative cache, LRU only needs to remember the order of elements in each set,

and in a zcache this can be achieved with timestamps, as explained in Section 3.3.5.

By convention, we give a higher rank r to blocks with a higher preference to be evicted. In a cache with B blocks, $r \in [0, ..., B-1]$. To make the rest of the analysis independent of cache size, we define a block's *eviction priority* to be its rank normalized to $[0, 1]$, i.e., $e = r/(B-1)$.

Associativity distribution: We define the associativity distribution as the *probability distribution* of the eviction priorities of evicted blocks. In a fully-associative cache, we would always evict the block with $e = 1.0$. However, most cache designs examine only a small subset of the blocks in an eviction, so they select blocks with lower eviction priorities. In general, the more skewed the distribution is towards $e = 1.0$, the higher the associativity is. The associativity distribution characterizes the *quality* of the replacement decisions made by the cache in a way that is independent of the replacement policy. Note that this decouples how the array performs from ill-effects from the replacement policy. For example, a highly associative cache may always find replacement candidates with high eviction priorities, but if the replacement policy does a poor job in ranking the blocks, this may actually hurt performance.

3.4.2 Linking Associativity and Replacement Candidates

Defining associativity as a probability distribution lets us evaluate the quality of the replacement candidates, but is still dependent on workload and replacement policy. However, under certain general conditions this distribution can be characterized by a single number, the number of replacement candidates. This is the figure of merit that zcaches optimize for.

Uniformity assumption: If the cache array always returns R replacement candidates, and we treat the eviction priorities of these blocks as random variables E_i, assuming that they are (1) *uniformly distributed* in [0,1] and (2) statistically *independent* from each other, we can derive the associativity distribution. Since

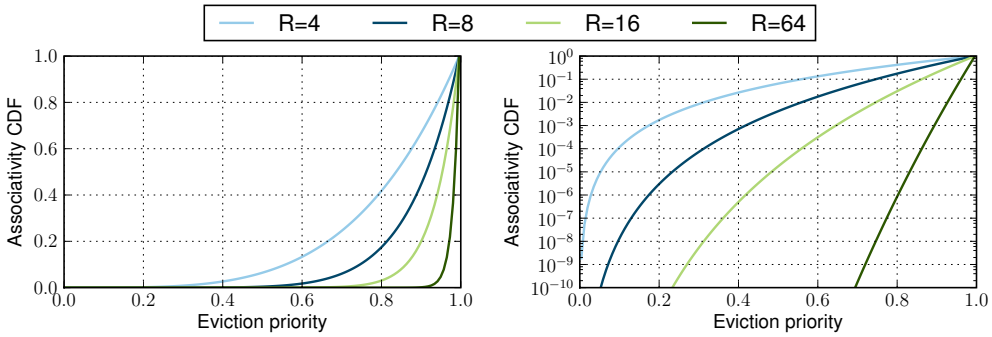


Figure 3.2: Associativity CDFs under the uniformity assumption ($F_A(x) = x^R, x \in [0,1]$) for $R = 4, 8, 16, 64$ replacement candidates, in linear and logarithmic scales.

$E_1, ..., E_R \sim U[0,1], i.i.d$, the cumulative distribution function (CDF) of each eviction priority is $F_{E_i}(x) = Prob(E_i \leq x) = x, x \in [0,1]$[1]. The associativity is the random variable $A = \max\{E_1, ..., E_R\}$, and its CDF is:

$$\begin{aligned} F_A(x) &= Prob(A \leq x) = Prob(E_1 \leq x \wedge ... \wedge E_R \leq x) \\ &= Prob(E_i \leq x)^R = x^R, x \in [0,1] \end{aligned} \tag{3.1}$$

Therefore, under this *uniformity assumption*, the associativity distribution only depends on R, the number of replacement candidates. Figure 3.2 shows example CDFs of the associativity distribution, in linear and semi-log scales, with each line representing a different number of replacement candidates. The higher the number of replacement candidates, the more skewed towards 1.0 the associativity distribution becomes. Also, evictions of blocks with a low eviction priority quickly become very rare. For example, for 16 replacement candidates, the probability of evicting a block with $e < 0.4$ is 10^{-6}.

Random candidates cache: The uniformity assumption makes it simple to characterize associativity, but it is not met in general by real cache designs. However,

[1]Note that we are treating E_i as *continuous* random variables, even though they are *discrete* (normalized ranks with one of B equally probable values in $[0,1]$). We do this to achieve results that are independent of cache size B. Results are the same for the discretized version of these equations.

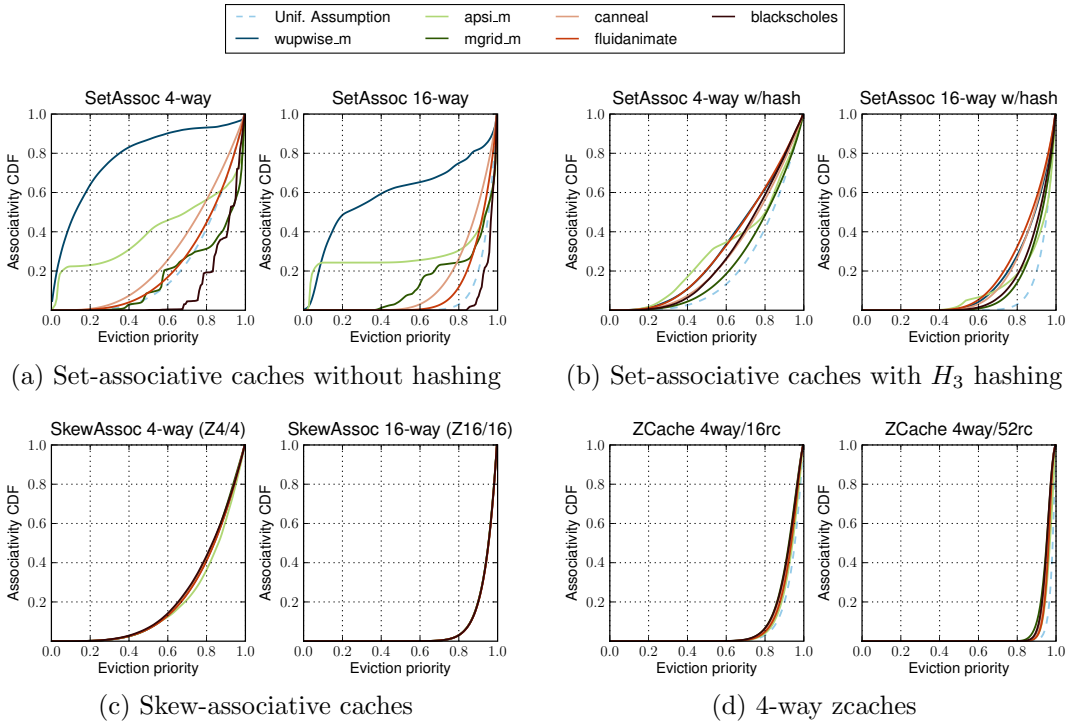
a cache array that returns n randomly selected replacement candidates (with repetition) from all the blocks in the cache always achieves these associativity curves *perfectly*. Each E_i is uniformly distributed because it is an unbiased *random sampling* of one of the B possible values of a rank, and since different selections are done independently, the E_i are independent as well. We simulated this cache design with tens of real workloads, under several configurations and replacement policies, and obtained associativity distributions as shown in Figure 3.2, experimentally validating the previous derivation.

Although this *random candidates* cache design is unrealistic, it reveals a sufficient condition to achieve the uniformity assumption: the more randomized the replacement candidates, the better a cache will match the uniformity assumption.

3.4.3 Associativity Measurements of Real Caches

Our analytical framework implies that the number of replacement candidates is the key metric in determining associativity. We now evaluate whether this is the case using real cache designs.

Set-associative caches: Figure 3.3a shows the associativity distributions for 8MB L2 set-associative caches of 4 and 16 ways, using an LRU replacement policy. The details on system configuration and methodology can be found in Section 3.5. Each of the 6 solid lines represents a different benchmark, from a representative selection of PARSEC and SPECOMP applications. The single dotted line per graph plots the associativity distribution under the uniformity assumption, which is independent of the workload. We see that the distributions differ significantly from the uniformity assumption. Two workloads (wupwise and apsi) do significantly worse, with the CDF rapidly climbing towards 1.0. For example, in wupwise, 60% of the evictions happen to blocks with $\leq 20\%$ eviction priority. Others (mgrid, canneal and fluidanimate) have sensibly worse associativity, and only one benchmark (blackscholes) outperforms the uniformity assumption. These differences are not surprising: replacement candidates all come from the same small set, thwarting independence, and locality of reference will skew eviction priorities towards lower values, breaking the assumption of an



(a) Set-associative caches without hashing

(b) Set-associative caches with H_3 hashing

(c) Skew-associative caches

(d) 4-way zcaches

Figure 3.3: Associativity distributions for selected PARSEC and SPECOMP workloads using different types of caches.

uniform distribution.

We can improve associativity with hashing. Figure 3.3b shows the associativity distributions of set-associative caches indexed by an H_3 hash of the block address. Associativity distributions generally improve, but some hot-spots remain, and all workloads now perform sensibly worse than the uniformity assumption case.

Skew-associative caches and zcaches: Figure 3.3c shows the associativity distributions of 4 and 16-way skew-associative caches. As we can see, skew-associative caches closely match the uniformity assumption on all workloads. These results provide an analytical foundation to the previous empirical observations that skew-associative caches "improve performance predictability" [20].

Figure 3.3d shows the associativity of 4-way zcaches with 2 and 3 levels of replacement candidates. We also observe a close match to the uniformity assumption.

This is expected, since replacement candidates are even more randomized: n^{th}-level candidates depend on the addresses of the $(n-1)^{th}$-level candidates, making the set of positions checked varying with cache contents.

In conclusion, both skew-associative caches and zcaches match the uniformity assumption in practice. Hence, their associativity is directly linked to the number of candidates examined on replacement. Although the graphs only show a small set of applications for clarity, results with other workloads and replacement policies are essentially identical. The small differences observed between applications decrease by either increasing the number of ways (and hash functions) or improving the quality of hash functions (the same experiments using more complex SHA-1 hash functions instead of H_3 yield distributions identical to the uniformity assumption).

Overall, our analysis framework reveals two main results:

1. In zcaches, associativity is determined by the *number of replacement candidates*, and not the number of ways, essentially *decoupling ways and associativity.*
2. When using an equal number of replacement candidates, zcaches empirically show better associativity than set-associative caches for most applications.

In the next two chapters, we will leverage the fact that zcaches match the uniformity assumption to implement scalable techniques that provide QoS in the cache hierarchy with minimal, controlled overprovisioning.

3.5 Experimental Methodology

Infrastructure: We perform microarchitectural, execution-driven simulation by developing and using zsim, an x86-64 simulator based on Pin [110]. We use McPAT [108] to obtain comprehensive timing, area and energy estimations for the CMPs we model, and use CACTI 6.5 [116] for more detailed cache area, power and timing models. We use 32nm ITRS models, with a high-performance process for all the components of the chip except the L2 cache, which uses a low-leakage process.

System: We simulate a 32-core CMP, with in-order x86 cores modeled after the Atom processor [62]. The system has a 2-level cache hierarchy, with a fully shared

Cores	32 cores, x86-64 ISA, in-order, IPC=1 except on memory accesses, 2 GHz
L1 caches	32 KB, 4-way set associative, split D/I, 1-cycle latency
L2 cache	8 MB NUCA , 8 banks, 1 MB bank, shared, inclusive, MESI directory coherence, 4-cycle average L1-to-L2-bank latency, 6–11-cycle L2 bank latency
MCU	4 memory controllers, 200 cycles zero-load latency, 64 GB/s peak memory BW

Table 3.1: Main characteristics of the simulated CMPs. The latencies assume a 32 nm process at 2GHz.

L2 cache. Table 3.1 shows the details of the system. On 32nm, this CMP requires about $220mm^2$ and has a TDP of around 90W at 2GHz, both reasonable budgets.

Workloads: We use a variety of multithreaded and multiprogrammed benchmarks: 6 PARSEC [16] applications (blackscholes, canneal, fluidanimate, freqmine, streamcluster and swaptions), 10 SPEC OMP2001 benchmarks (all except galgel, which gcc cannot compile) and 26 SPEC CPU2006 programs (all except dealII, tonto and wrf, which we could not compile). For multiprogrammed runs, we run different instances of the same single-threaded CPU2006 application on each core, plus 30 random CPU2006 workload combinations (choosing 32 workloads each time, with repetitions allowed). These make a total of 72 workloads. All applications are run with their reference (maximum size) input sets. For multithreaded workloads, we fast-forward into the parallel region and run the first 10 billion instructions. Since synchronization can skew IPC results for multithreaded workloads [7], we do not count instructions in synchronization routines (locks, barriers, etc.) to determine when to stop execution, but we do include them in energy calculations. For multiprogrammed workloads, we follow standard methodology from prior work [85]: we fast-forward 20 billion instructions for each process, simulate until all the threads have executed at least 256 million instructions, and only take the first 256 million instructions of each thread into account for IPC computations.

3.6 Evaluation

ZCache can be used with any design that requires high associativity at low overheads in terms of area, hit time, and hit energy. In this chapter, we evaluate zcache as a

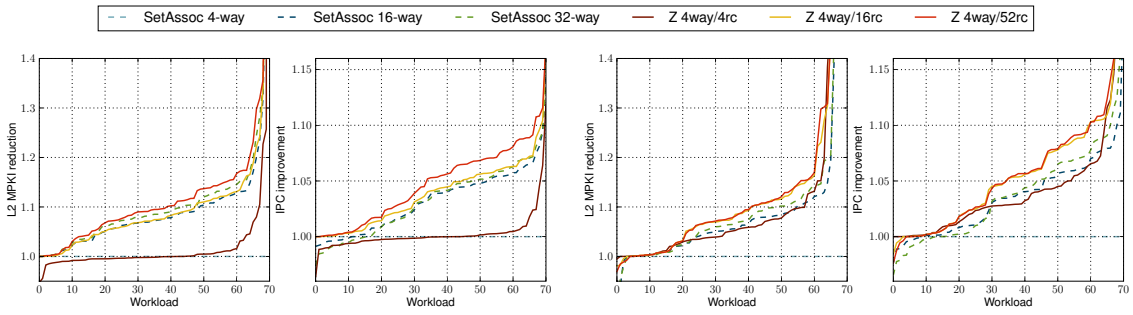
Cache	Serial lookups			Parallel lookups				
	Bank lat	Bank E/hit	Bank E/miss	Bank lat	Bank E/hit	Bank E/miss	L2 area	L2 leakage
SA 4-way	4.14 ns	0.61 nJ	1.26 nJ	2.91 ns	0.71 nJ	1.42 nJ	42.3 mm^2	535 mW
SA 8-way	4.41 ns	0.75 nJ	1.57 nJ	3.18 ns	0.99 nJ	1.88 nJ	45.1 mm^2	536 mW
SA 16-way	4.74 ns	0.88 nJ	1.87 nJ	3.51 ns	1.42 nJ	2.46 nJ	46.4 mm^2	561 mW
SA 32-way	5.05 ns	1.23 nJ	2.66 nJ	3.82 ns	2.34 nJ	3.82 nJ	51.9 mm^2	588 mW
ZCache 4/16	4.14 ns	0.62 nJ	2.28 nJ	2.91 ns	0.72 nJ	2.44 nJ	42.3 mm^2	535 mW
ZCache 4/52	4.14 ns	0.62 nJ	3.47 nJ	2.91 ns	0.72 nJ	3.63 nJ	42.3 mm^2	535 mW

Table 3.2: Area, power and latency of 8MB, 8-banked L2 caches with different organizations.

last-level cache in a 32-node CMP. We first quantify the area, energy and latency advantages of zcaches versus set-associative caches with similar associativity, then compare the performance and system-wide energy over our set of workloads.

3.6.1 Cache Costs

Table 3.2 shows the timing, area and power requirements of both set-associative caches and zcaches with varying associativities. We use CACTI's models to obtain these numbers. Tag and data arrays are designed separately by doing a full design space exploration and choosing the design that minimizes area×delay×power. Arrays are sub-banked, and both the address and data ports are implemented using H-trees. We show results for both serial and parallel-lookup caches. In serial caches, tag and data arrays are accessed sequentially, saving energy at the expense of delay. In parallel caches, both tag and data accesses are initiated in parallel. When the tag read resolves the appropriate way, it propagates a way-select signal to the data array, which selects and propagates the correct output. This parallelizes most of the tag and data accesses while avoiding an exceedingly wide data array port. For zcaches, we explore designs with two and three-level walks. We denote zcaches with "W/R", indicating the number of ways and replacement candidates, respectively. For example, a 4/16 zcache has 4 ways and 16 replacement candidates per eviction (obtained from a two-level walk).



(a) OPT replacement

(b) LRU replacement

Figure 3.4: L2 MPKI and IPC improvements for all workloads, over a 4-way set-associative with hashing baseline.

Table 3.2 shows that increasing the number of ways beyond 8 starts imposing significant area, latency and energy overheads. For example, a 32-way cache with serial lookups has 1.22× the area, 1.23× the hit latency and 2× the hit energy of a 4-way cache (for parallel lookups, hit latency is 1.32× and hit energy is 3.3×). This is logical, since a 32-way cache reads 4× more tag bits than data bits per lookup, the tag array has a much wider port, and the critical path is longer (slower tag array, more comparators). For zcaches, however, area, hit latency and hit energy grow with the number of ways, but not with the number of replacement candidates. This comes at the expense of increasing energy per miss, which, however, is still similar to set-associative caches with the same associativity. For example, a serial-lookup zcache 4/52 has almost twice the associativity of a 32-way set-associative cache at 1.3× higher energy per miss, but retains the 2× lower hit energy and 1.23× lower access latency of a 4-way cache.

3.6.2 Performance

Figure 3.4 shows the improvements in both L2 misses per thousand instructions (MPKI) and IPC for all workloads, using both OPT and LRU replacement policies. Each line represents the improvement of a different cache design over a baseline 4-way set-associative cache with H_3 hashing. Caches without hashing perform significantly

worse (even at high associativities), so we do not consider them here. Serial-lookup caches are used in all cases. For each line, workloads (in the x-axis) are sorted according to the improvement achieved, so each line is monotonically increasing. Fractional improvements are given (e.g., a L2 MPKI reduction of 1.2 means 1.2× lower MPKI than the baseline).

OPT: Figure 3.4a shows the effects of using OPT replacement (i.e., evicting the candidate reused furthest). OPT simulations are run in trace-driven mode. Although OPT is unrealistic, it removes ill-effects from the replacement policy (where e.g., increasing associativity degrades performance), allowing us to decouple replacement policy issues from associativity effects[2]. Note that these numbers do not necessarily show maximum improvements from increasing associativity, as other replacement policies may be more sensitive to associativity changes. In terms of misses, higher associativities always improve MPKI, and designs with the same associativity have practically the same improvements (e.g., 16-way set-associative vs Z4/16). However, for set-associative caches, these improvements in MPKI do not always translate to IPC, due to the additional access latency (1 extra cycle for 16-way, 2 cycles for 32-way). For example, the 32-way set-associative design performs worse than the 4-way design on 15 workloads (which have a large number of L1 misses, but few L2 misses), and performs worse than the 16-way design on half of the workloads (36). In contrast, zcaches do not suffer from increased access latency, sensibly improving IPC with associativity for all workloads (e.g., a Z4/52 improves IPC by up to 16% over the baseline).

LRU: Figure 3.4b compares cache designs when using LRU. Associativity improves MPKI for all but 3 workloads, and both MPKI and IPC improvements are significant (e.g., a Z4/52 reduces L2 misses by up to 2.1× and improves performance by up to 25% over a 4-way set-associative cache). With LRU, the difference between Z4/16 and Z4/52 designs is lower than with OPT, however they significantly outperform both the baseline and the Z4/4 (skew-associative) design.

[2]In caches with interference across sets, like skew-associative caches and zcaches, OPT is not actually optimal, but it is a good heuristic. In fact, in these caches the optimal replacement policy is NP-hard [24]

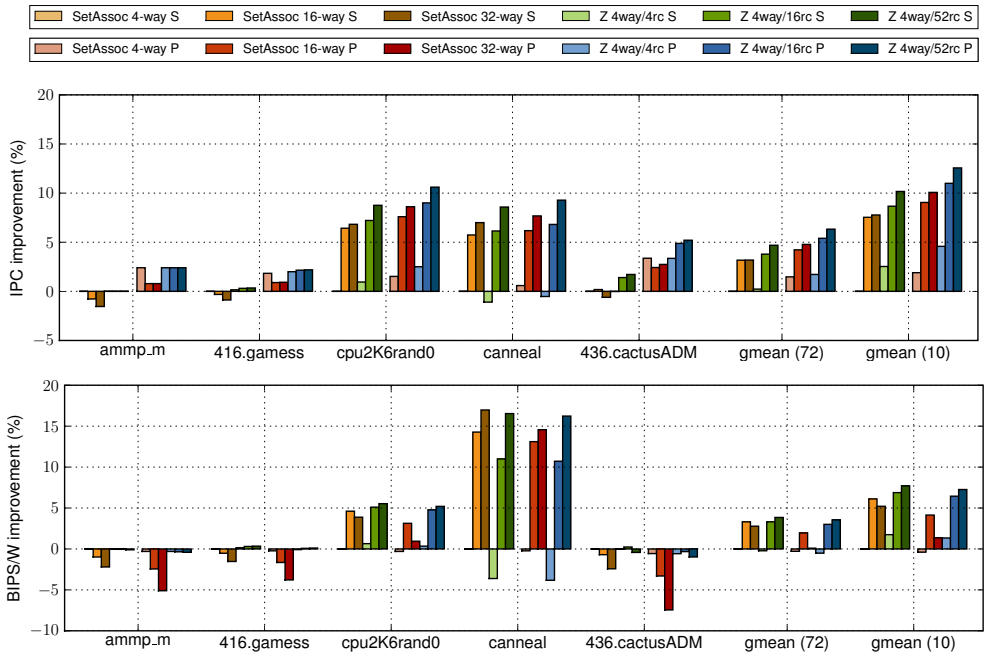
3.6.3 Serial vs Parallel-Lookup Caches

Figure 3.5 shows the performance and *system-wide* energy efficiency when using serial and parallel-lookup caches, under both OPT and LRU replacement policies. Results are normalized to a serial-lookup, 4-way set-associative cache with H_3 hashing. Each graph shows improvements on five representative applications, as well as the geometric means of both all 72 workloads and the 10 workloads with the highest L2 MPKI.

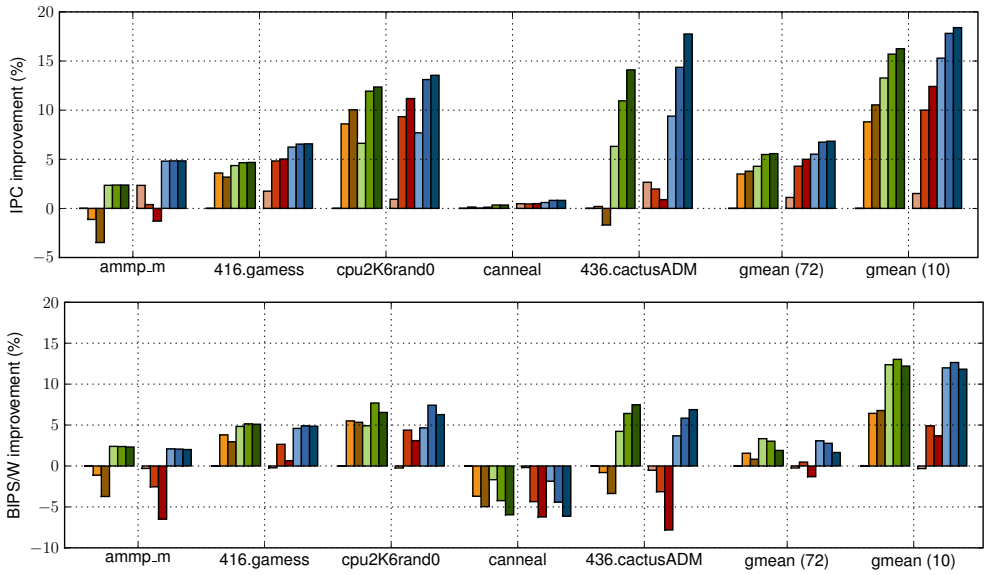
We can distinguish three types of applications: a few benchmarks, like blackscholes or freqmine, have low L1 miss rates, and are insensitive to the L2's organization. Other applications, like ammp and gamess, have frequent L2 hits but infrequent L2 misses. These workloads are sensitive to hit latency, so parallel-lookup caches provide higher performance gains than increasing associativity (e.g., a 3% IPC improvement on gamess vs serial-lookup caches). In fact, increasing associativity in set-associative caches reduces performance due to higher hit latencies, while highly-associative zcaches do not degrade performance. Finally, workloads like cpu2K6rand0, canneal, and cactusADM have frequent L2 misses. These applications are often sensitive to associativity, and a highly-associative cache improves performance (by reducing L2 MPKI) more than reducing access time (e.g., in cactusADM with LRU, going from Z4/4 to Z4/52 improves IPC by 9%, while going from serial to parallel-lookup improves IPC by 3%).

In terms of energy efficiency, set-associative caches and zcaches show different behaviors when increasing associativity. Because hit energy increases steeply with the number of ways in parallel-lookup caches, 16 and 32-way set-associative caches often achieve lower energy efficiency than serial-lookup caches (e.g., up to 8% lower BIPS/W in cactusADM). In contrast, serial and parallel-lookup zcaches achieve practically the same energy efficiency on most workloads, due to their similarly low access and miss energies. In conclusion, zcaches enable highly-associative, energy-efficient parallel-lookup caches.

Overall, zcaches offer both the best performance and energy efficiency. For example, under LRU, when considering all 72 workloads, a parallel-lookup zcache 4/52 improves IPC by 7% and BIPS/W by 3% over the 4-way baseline. Over the subset of the 10 most L2 miss-intensive workloads, a zcache 4/52 improves IPC by 18% and



(a) OPT replacement



(b) LRU replacement

Figure 3.5: IPC and energy efficiency (BIPS/W) improvements for serial and parallel-lookup caches, over a serial-lookup 4-way set-associative with hashing baseline. Each graph shows improvements for 5 representative workloads, plus the geometric mean over both all 72 workloads and the 10 workloads with the highest L2 MPKI.

energy efficiency by 13% over the 4-way baseline, and obtains 7% higher performance and 10% better energy efficiency than a 32-way set-associative cache.

3.6.4 Array Bandwidth

Since zcaches perform multiple tag lookups on a miss, it is worth examining whether these additional lookups can saturate bandwidth. Of the 72 workloads, the maximum average load per bank is 15.2% (i.e., 0.152 core accesses/cycle/L2 bank). However, as L2 misses increase, average load decreases: at 0.005 misses/cycle/bank, average load is 0.035 accesses/cycle/bank, and total load on the tag array for a Z4/52 cache is 0.092 tag accesses/cycle/bank. In other words, as L2 misses increase, bandwidth pressure on the L2 decreases; the system is self-throttling. ZCaches use this spare tag bandwidth to improve associativity. Ultimately, even for high-MLP architectures, the load on the tag arrays is limited by main memory bandwidth, which is more than an order of magnitude smaller than the maximum L2 tag bandwidth and much harder to scale.

3.7 Additional Related Work

ZCache is inspired by cuckoo hashing, a technique to build space-efficient hash tables proposed by Pagh and Rodler [124]. The original design uses two hash functions to index the hash table, so each lookup needs to check two locations. On an insertion, if both possible locations are occupied, the incoming item replaces one of them at random, and the replaced block is reinserted. This is repeated until either an empty location is found or, if a limit number of retries is reached, elements are rehashed into a larger array. Though cuckoo hashing has been mostly studied as a technique for software hash tables, hardware variants have been proposed to implement lookup tables in IP routers [51]. For additional references, Mitzenmacher has a survey on recent research in cuckoo hashing [114].

Both high associativity and a good replacement policy are necessary to improve

cache performance. The growing importance of cache performance has sparked research into alternative policies that outperform LRU [36, 83, 85, 163]. The increasing importance of on-chip wire delay has also motivated research in non-uniform cache architectures (NUCA) [95]. Some NUCA designs such as NuRAPID [43] use indirection to enhance the flexibility of NUCA placement and reduce access latency instead of increasing associativity.

3.8 Summary

This chapter has presented zcache, a cache design that enables high associativity with a small number of ways and provides analytical, workload-independent guarantees on associativity. ZCache uses a different hash function per way to enable an arbitrarily large number of replacement candidates on a miss. To evaluate zcache's associativity, we have developed an analytical framework to characterize and compare associativity. We use this framework to show that, for zcaches, associativity is determined by the number of replacement candidates, not the number of ways, hence decoupling ways and associativity. An evaluation using zcaches as the last-level cache in a CMP shows that they provide high associativity with low overheads in terms of area, hit time, and hit energy.

Chapter 4

Vantage: Scalable and Efficient Cache Partitioning

4.1 Introduction

Shared caches are pervasively used in CMPs, especially in the higher levels of the memory hierarchy, because they have significant utilization and efficiency benefits over private caches. However, when multiple applications share the CMP, they suffer from *interference* in shared caches. This causes large performance variations, precluding quality of service (QoS) guarantees, and can degrade cache utilization, hurting overall throughput. As we discussed in Section 2.2, cache partitioning can eliminate this interference, but previously proposed partitioning techniques have two main drawbacks: they are limited to a small number of coarse-grain partitions, so they are not scalable, and partitioning often degrades performance, imposing a trade-off between high performance or predictability and QoS.

In this chapter we present Vantage, a cache partitioning scheme that avoids the drawbacks of prior techniques. Vantage can maintain hundreds of partitions defined at cache line granularity, provides strict isolation among partitions, maintains high cache performance, and is simple to implement, requiring minimal overheads. Thanks to these features, Vantage enables performance isolation and quality of service in current and future large-scale CMPs, and can be used for several other purposes, such as

cache-pinning critical data, or implementing flexible local stores through application-controlled partitions. To this end, this chapter presents the following contributions:

1. We present the design of Vantage. Unlike other techniques, *Vantage is designed and fully characterized by accurate analytical models*. Vantage leverages the analytical associativity guarantees of zcaches (Section 3.4), which enable soft-pinning a large portion of the lines by simply modifying the replacement process. Vantage does not physically restrict line placement, side-stepping the problems of previous strict partitioning techniques ((Section 2.2.2), and leverages analytical models to provide strict guarantees on partition sizes and interference, independently of workload behavior. To provide these guarantees, Vantage partitions most of the cache, not all of it. Partitions can slightly outgrow their target allocations, but they borrow space from a small unpartitioned region of the cache, not from other partitions. Hence, Vantage eliminates destructive interference between partitions. Vantage maintains partition sizes by matching the average rates at which lines enter and leave each partition. We prove that by controlling partition sizes this way, the amount of cache space that has to be left unpartitioned for Vantage to work well is both small (e.g., around 5-15% in a 4-way zcache) and *independent of the number of partitions or their sizes*. Therefore, Vantage is *scalable*. Vantage also works with conventional set-associative caches, although with slightly reduced performance and weaker guarantees.
2. While the conceptual techniques that Vantage relies on provide strong guarantees, implementing them directly would be complex. We propose a practical design that relies on negative feedback to control partition sizes in a way that maintains the guarantees of the analytical models without their complexity. Our design just requires adding few bits to each tag (e.g., 6 bits to support 32 partitions) and simple modifications to the cache controller, which only needs to track about 256 bits of state per partition, and a few narrow adders and comparators for its control logic. On an 8 MB last-level cache with 32 partitions, Vantage adds a 1.1% state overhead overall.
3. We evaluate Vantage by simulating a large variety of multiprogrammed workloads on both 4-core and 32-core CMPs. We compare it to way-partitioning [42] and

Scheme	Scalable & fine-grain	Maintains associativity	Efficient resizing	Strict sizes & isolation	Indep. of repl. policy	HW cost	Partitions whole cache
Way-partitioning [42, 132]	No	No	Yes	Yes	Yes	Low	Yes
Set-partitioning [132, 155]	No	Yes	No	Yes	Yes	High	Yes
Page coloring [109]	No	Yes	No	Yes	Yes	None (SW)	Yes
Ins/repl policy-based [82, 162, 163]	Some-times	Some-times	Yes	No	No	Low	Yes
Vantage	Yes	Yes	Yes	Yes	Yes	Low	No (most)

Table 4.1: Classification of partitioning schemes.

PIPP [163] using utility-based cache partitioning (UCP) [128] as the allocation policy. Vantage significantly improves the performance of UCP on the 4-core system (up to 40%), but results are most striking on the 32-core system: while using either way-partitioning or PIPP to partition a 64-way cache almost always degrades performance due to the large loss of associativity, Vantage is able to deliver similar performance improvements as in the 4-core system, maintaining 32 fine-grain, highly-associative partitions *using a 4-way cache* (i.e., 16 times fewer ways). Additional simulation results show that Vantage achieves the benefits and bounds predicted by the analytical models.

4.2 Background on Cache Partitioning

Partitioning requires an *allocation policy* to decide the number and sizes of partitions, and a *partitioning scheme* to *enforce* them. In this work we focus on the latter. Table 4.1 summarizes the differences between current approaches, which we review in this section. Broadly, there are two approaches to partition a cache:

Strict partitioning by restricting line placement: Schemes with strict partitioning guarantees rely on restricting the locations where a line can be placed depending on its partition. Way-partitioning or column caching [42] divides the cache by ways, restricting fills from each partition to its assigned subset of ways. Way-partitioning is simple, but has several problems: partitions are coarsely sized (in multiples of way size), the number of partitions is limited by the number of ways, and the associativity of each partition is proportional to its way count, imposing a trade-off between isolation and partition performance. For way-partitioning to work well, the number of ways should be significantly larger than the number of partitions, so this scheme does not scale to large partition counts.

To avoid losing associativity, the cache can be partitioned by sets instead of ways, as proposed by one flavor of reconfigurable caches [132] and molecular caches [155]. However, these approaches require configurable decoders or a significant redesign of cache arrays, and must do scrubbing, i.e., flushing or moving data when resizing partitions. Most importantly, this scheme will only work when we have fully disjoint address spaces, which is not true in most cases. Even different applications operating on separate address spaces share library code and OS code and data. A different approach to partition through placement restriction is to leverage virtual memory, using page coloring to constrain the physical pages of a process to map to a portion of the cache sets [109]. While this scheme does not require hardware support, it is limited to coarse-grain partition sizes (multiples of page size×cache ways), precludes the use of superpages, does not work on caches that are indexed using hashing (common in modern processors [141]), and repartitioning requires costly recoloring (i.e., copying) of physical pages, so it must be done infrequently [109].

Soft partitioning by controlling insertion and/or replacement: Alternatively, a cache can be partitioned approximately by modifying the allocation or replacement policies. These schemes avoid some of the issues of restricting line placement, but provide only limited control over partition sizes and inter-partition interference. They are useful for partitioning policies that can work with approximate

partitioning, but not for uses that require stricter guarantees. In selective cache allocations [82] each partition is assigned a probability p, and incoming lines from that partition are inserted with probability p or discarded (self-replaced) with probability $1 - p$. In decay-based replacement policies, lines from different partitions age at different rates; adjusting the rates provides some control over partition sizes [162]. Promotion-insertion pseudo-partitioning (PIPP) [163] assigns each partition a different insertion position in the LRU chain and slowly promotes lines on hits (e.g., promoting $\simeq 1$ position per hit instead of moving the line to the head of the LRU chain). With an additional mechanism to restrict cache pollution of thrashing applications, PIPP approximately attains the desired partition sizes. PIPP is co-designed to work with UCP as the allocation policy, and may not work correctly with other policies. Finally, as we will see in Section 4.6, PIPP's partitioning scheme does not scale with the number of partitions.

4.3 Vantage Techniques

4.3.1 Overview

Vantage relies on caches with high associativity and good hashing that meet the uniformity assumption (Section 3.4), such as zcaches. These caches provide high, predictable associativity regardless of the workload, and thus can keep a large portion of the lines effectively pinned in the cache (see Section 4.3.2).

Vantage does not physically restrict line placement: lines from all partitions share the cache. It enforces partition sizes at replacement time. On each replacement, Vantage needs to evict one line from a set of replacement candidates. In a partitioned cache, this set may include good candidates from other partitions (i.e., lines that the owning partition would have to evict anyway). To strictly enforce partition sizes, we should always evict a candidate from the same partition as the incoming line. However, this does not scale with the number of partitions, as the portion of candidates from that specific partition will be increasingly small with more partitions. For example, a 16-way set-associative cache has 16 replacement candidates to choose from

when unpartitioned, but only 2 when it is evenly divided in 8 partitions. The core idea behind Vantage is to relax this restriction, imposing only that the rates of insertions and evictions from each partition *match on average*. Since Vantage dynamically adjusts how to select candidates based on the insertion rate of each partition, we call this technique *churn-based management* (Section 4.3.4).

Unfortunately, churn-based management alone has several drawbacks: it allows interference across partitions (as choosing a candidate from another partition means taking space away from that partition and giving it to the one that caused the miss), makes it hard to provide strong guarantees on partition sizes, and requires a complex controller. To solve these issues, we partition *most* of the cache rather than all of it. We divide cache space into a managed region and a small unmanaged region (e.g., 15% of the cache), and partition only the managed region. Partitions can slightly outgrow their target allocations, borrowing space from the unmanaged region instead of from each other. This *managed-unmanaged region division* (Section 4.3.3) solves all interference issues, allows for a simple controller design, and significantly increases the associativity on the managed region.

Vantage's control scheme is derived from statistical analysis rather than empirical observation. It achieves *provable, strong guarantees*, namely, it eliminates inter-partition interference, provides precise control of partition sizes, and maintains high partition associativities, regardless of the number of partitions or the workload.

4.3.2 Caches with High Associativity

Vantage relies on highly associative caches that meet the uniformity assumption (Section 3.4) to provide analytical guarantees. ZCache is well-suited for this purpose, since it can provide high associativity efficiently. However, skew-associative caches (which zcache generalizes) can also be used. These caches exhibit two very useful properties for partitioning: they can restrict evictions to a specific portion of lines by simply controlling the replacement policy, and provide high associativity independently of the workload's access pattern.

For caches that meet the uniformity assumption, the associativity distribution can

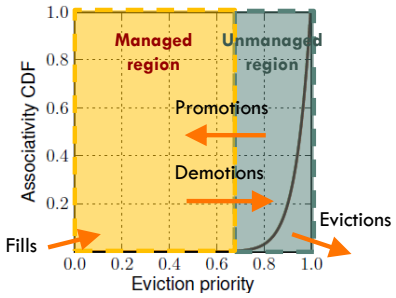
be derived analytically (Section 3.4.2), and is given by Equation 3.1. Figure 3.2 plots this distribution for a varying number of replacement candidates. In particular, note that with a large number of replacement candidates, the probability of evicting lines with a low eviction priority quickly becomes negligible. For example, with $R = 64$, the probability of evicting a line with eviction priority $e < 0.8$ is $F_A(0.8) = 10^{-6}$. Hence, by simply controlling how lines are ranked, we can guarantee that they will be kept in the cache with a very high probability.

Vantage assumes that the underlying cache design meets the uniformity assumption $F_A(x)$. However, Vantage is not limited to zcaches and skew-associative caches. In Section 4.6 we show that Vantage can be used with hashed set-associative caches, although at higher cost (more ways) and with a slight loss of performance and analytical guarantees.

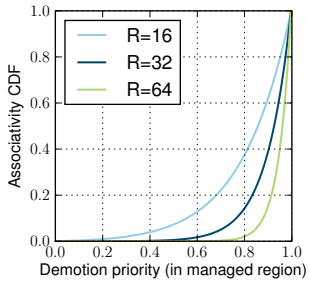
Assumptions: For the rest of this section, we make two assumptions in our analysis. First, we assume that the replacement candidates on each eviction are independent and uniformly distributed. Although this is not strictly the case, it is close enough (Section 3.4.3) that our models are accurate in practice, as we will see in Section 4.6. Second, we assume that, on each replacement, we know the eviction priority of every candidate, as given by the replacement policy. While tracking eviction priorities would be very expensive in practice, Section 4.4 shows that we can achieve similar results with a much simpler scheme.

4.3.3 Managed-Unmanaged Region Division

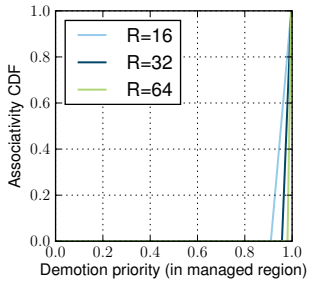
We divide the cache in two logical regions: a *managed* and an *unmanaged* region. This division is done by simply tagging each line as either managed or unmanaged, and region sizes are set by controlling the flow of lines between the two regions. A *base replacement policy* (e.g., LRU) ranks lines as in an undivided cache, oblivious to the existence of the two regions. On an eviction, lines in the unmanaged region are always prioritized for eviction over managed lines. The unmanaged region is sized so that it captures most evictions, making evictions in the managed region negligible.



(a) Managed-unmanaged region division for a $R = 16$ replacement candidates cache, and flows between regions.



(b) Associativity CDF in the managed region when doing *exactly* one demotion per eviction.



(c) Associativity CDF in the managed region when doing one demotion per eviction *on average*.

Figure 4.1: Managed-unmanaged region division: setup, flows and associativity in the managed region (assuming 30% of the cache is unmanaged).

Figure 4.1a illustrates this setup. It shows the associativity distribution of a cache with $R = 16$ candidates, divided in the managed and unmanaged regions, and the flows of lines between the two. To make evictions in the managed region negligible ($\simeq 10^{-3}$ probability), the unmanaged region is sized to 30% of the cache. Caches with $R > 16$ will require a smaller unmanaged region. Incoming lines are *inserted* in the managed region, eventually *demoted* to the unmanaged region, and either *evicted* from there, or *promoted* if they get a hit. Promotions and demotions do not physically move the line, just change its tag.

In a sense, the unmanaged region acts as a victim cache for the managed region. Evicting a line requires that it be demoted first (saving for the rare cases where we do not find a candidate from the unmanaged region). To keep the sizes of both regions constant, we would have to demote one line on each replacement and promotion. We denote the fraction of the cache devoted to the managed and unmanaged regions by m and u, respectively (e.g., in Figure 4.1a, $m = 0.7$ and $u = 0.3$). Ignoring the flow of promotions (which is typically small compared to the evictions), if we demote exactly one line on each replacement, the associativity distribution *for demotions* inside the

managed region is:

$$F_M(x) \cong \sum_{i=1}^{R-1} B(i,R) F_{A_i}(x) \tag{4.1}$$

where $B(i,R) = \binom{R}{i}(1-u)^i u^{R-i}$ is the probability that i of the R replacement candidates are in the managed region (a binomial distribution), and $F_{A_i}(x) = x^i$ is the nominal associativity distribution with i replacement candidates[1]. Figure 4.1b plots this distribution for various values of R.

To maintain the sizes of the two regions under control, however, it is not necessary to demote exactly one candidate per eviction. It suffices to demote one *on average*. For example, some evictions might not yield any candidates with high eviction priority from the managed region, while others might find two or more. By allowing demotions to work on the average case rather than being affected by the worst case, associativity increases significantly. In this case the controller only needs to select a threshold value, which we call the *aperture* (A), over which it will demote every candidate that it finds. For example, if $A = 0.05$, it will demote every candidate that is on the top 5% of eviction priorities (i.e., $e \geq 0.95$). Since, on average, $R \cdot m$ of the candidates are from the managed region, maintaining the sizes requires an aperture $A = \frac{1}{R \cdot m}$. The associativity distribution in the managed region is uniform $\sim U[1-A, 1]$, so the CDF is:

$$F_M(x) = \begin{cases} 0 & \text{if} \quad x < 1 - A \\ \frac{x-(1-A)}{A} & \text{if} \quad 1 - A \leq x \leq 1 \\ 1 & \text{if} \quad x > 1 \end{cases} \tag{4.2}$$

Figure 4.1c shows the associativity distributions for several values of R. By comparing Figure 4.1b and Figure 4.1c, we clearly see that demoting on the average significantly improves associativity. For example, with $R = 16$ candidates, demoting on the average only demotes lines with eviction priority $e > 0.9$. Meanwhile, when demoting always one line per eviction, 60% of the demotions will happen to lines with $e < 0.9$.

Overall, using the unmanaged region has several advantages. First, it enables

[1]This formula is approximate, because we ignore the cases $i = 0$ (no replacement candidates are from the managed region, hence none can be demoted), and $i = R$ (all the candidates are from the managed region, so we need to evict from the managed region rather than demote). Both cases have a negligible probability.

the controller to work on the average in the managed region, increasing associativity. Second, once we partition the managed region, partitions will borrow space from it instead of from each other, eliminating inter-partition interference. Third, it will make it practical to implement a Vantage controller (Section 4.4). While a portion of the cache must remain unpartitioned, this is typically a small percentage, e.g., 5-15% with $R = 52$ candidates (Section 4.4).

4.3.4 Churn-based Management

We now logically partition the managed *region*[2]. We have P *partitions* of *target sizes* $T_1, ..., T_P$, so that $\sum_{i=1}^{P} T_i = m$ (i.e., partition sizes are expressed as a fraction of the total cache size). These target sizes are given to Vantage by the allocation policy (e.g., UCP or software mechanisms). Partitions have *actual sizes* $S_1, ..., S_P$, and insertion rates, which we call *churns*, $C_1, ..., C_P$ (a partition's churn is measured in insertions per unit of time). Churn-based management keeps the actual size of each partition close to its target size by matching its demotion rate with its churn. It achieves this by controlling how demotions are done. Instead of having one aperture for the managed region, there is one aperture *per partition*, A_i. On each replacement, all the candidates below their partitions' apertures are demoted. Unlike way-partitioning, which achieves isolation by always evicting a line from the inserting partition, Vantage allows a partition's incoming line to demote others' lines. Vantage embraces interference and uses it to its advantage.

We now describe how churn-based management works on different cases. As in Section 4.3.3, we ignore the flow of promotions to simplify the analysis. Promotions are rare compared to insertions, hence we treat them as a small modeling error, addressed when implementing the controller (Section 4.4).

Partitions with similar behavior: The simplest case happens when partitions have both the same sizes (S_i) and churns (C_i). In this case, keeping all apertures equal, $A_i = \frac{1}{R \cdot m}$, will maintain their sizes. This is independent of how the base

[2]Note the distinction between regions and partitions: Vantage keeps two *regions*, managed and unmanaged, and divides the managed region in *partitions*.

replacement policy ranks candidates, as we are demoting from the bottom A_i portion from each partition. Furthermore, the aperture is independent from the number of partitions: *Vantage retains the same associativity as if the cache was unpartitioned.*

Partitions with different behaviors: When partitions have different sizes and/or churns, apertures need to accommodate for this. A partition with a higher churn than the average will need a larger aperture, as we need to demote its lines at a higher frequency; and a partition that is smaller in size than the average will also need a larger aperture, because replacement candidates from that partition will be found more rarely.

Overall, partitions with a larger churn and/or a smaller size than the average will have a larger aperture, and partitions with a smaller churn and/or a larger size than the average will have a smaller aperture. For example, consider a case with 4 equally sized partitions ($S_1 = S_2 = S_3 = S_4$), where the first partition has twice the churn as the others ($C_1 = 2C_2$, $C_2 = C_3 = C_4$). The cache examines $R = 16$ replacement candidates per eviction, and the managed region takes $m = 62.5\%$ of the cache. On each replacement, $R \cdot m = 16 \cdot 0.625 = 10$ candidates are in the managed region on average. To maintain the partitions' sizes, on average, for every 5 demotions, 2 should be done from partition 1, and 1 demotion from each of partitions 2, 3 and 4. Every 5 demotions, Vantage gets $5 \cdot 10 = 50$ candidates from the managed region on average, $50/4 = 12.5$ candidates per partition since they are equally sized. Therefore, the apertures need to be $A_1 = 2/12.5 = 16\%$ for partition 1, and $A_2 = A_3 = A_4 = 1/12.5 = 8\%$ for the other partitions. Hence, partitions with disparate churns or sizes cause associativity to be unevenly distributed.

In general, when we have partitions with different sizes S_i and churns C_i, we can derive the aperture of each partition. Out of the $R \cdot m$ replacement candidates per demotion that fall in the managed region, a fraction $\frac{S_i}{\sum_{k=1}^{P} S_k}$ are from partition i, and we need to demote lines at a fractional rate of $\frac{C_i}{\sum_{k=1}^{P} C_k}$ in this partition. Therefore,

$$A_i = \frac{C_i}{\sum_{k=1}^{P} C_k} \frac{\sum_{k=1}^{P} S_k}{S_i} \frac{1}{R \cdot m} \tag{4.3}$$

Stability: Depending on the sizes and churns of the partitions, simply adjusting their apertures may not be enough to maintain their sizes. Even if we are willing to sacrifice associativity by allowing the aperture to reach up to 1.0 (demoting every candidate from this partition), a partition with a large C_i/S_i ratio may require a larger aperture. Since it is undesirable to completely sacrifice associativity to maintain partition sizes, we set a *maximum aperture* A_{max}. If using Equation 4.3 yields an aperture larger than A_{max}, we have three options. First, we can do nothing and let the partition grow beyond its target allocation, borrowing space from the unmanaged region. Second, we can allow low-churn/size $\rightarrow$ high-churn/size partition interference by inserting its lines in the unmanaged region (throttling its churn). Third, we can allow high-churn/size $\rightarrow$ low-churn/size partition interference by reducing the size of one or more low-churn partitions and allocating that space to the high-churn partition until its aperture is lower than A_{max}.

Doing nothing could lead, in principle, to borrowing too much space from the unmanaged region, making it too small and leading to frequent forced evictions from the managed region, breaking our scheme. However, this is not the case if we allow for some extra slack when sizing the unmanaged region. Consider what happens when several partitions cannot match their minimum sizes. Specifically, partitions $1, ..., Q$ ($Q < P$) have very small sizes (e.g., 1 line each) and high churns. Each partition will grow until it is large enough that its C_i/S_i ratio can be handled with aperture A_{max}. This *minimum stable size* is:

$$MSS_j = \frac{C_j}{\sum_{k=1}^{P} C_k} \frac{\sum_{k=1}^{P} S_k}{A_{max} \cdot R \cdot m}, \forall j \in \{1, ..., Q\} \tag{4.4}$$

(obtained from Equation 4.3, with $S_j = MSS_j$ and $A_j = A_{max}$). Additionally, in the worst case, all other partitions ($Q+1, ..., P$) have zero churn, so $\sum_{k=1}^{P} C_k = \sum_{k=1}^{Q} C_k$. In this case, the total space borrowed from the unmanaged region is:

$$\sum_{j=1}^{Q} MSS_j = \frac{\sum_{j=1}^{Q} C_j}{\sum_{k=1}^{P} C_k} \frac{\sum_{k=1}^{P} S_k}{A_{max} \cdot R \cdot m} = \frac{\sum_{k=1}^{P} S_k}{A_{max} \cdot R \cdot m} \tag{4.5}$$

and assuming $\sum_{k=1}^{P} S_k \cong m$, $\sum_{j=1}^{Q} MSS_j \cong 1/(A_{max}R)$. For the exact derivation, $\sum_{k=1}^{P} S_k = \sum_{k=1}^{P} T_k + \sum_{j=1}^{Q} MSS_j$, and the target sizes achieve $\sum_{k=1}^{P} T_k = m$. By substituting on the previous equation, $\sum_{j=1}^{Q} MSS_j = 1/(A_{max}R-1/m)$. For any reasonable values of A_{max}, R and m, $A_{max}R \gg 1/m$, and therefore $\sum_{j=1}^{Q} MSS_j \cong 1/(A_{max}R)$ is a fine approximation. Hence, sizing the unmanaged region with an extra $1/(A_{max}R)$ of the cache guarantees that the scheme maintains the desired number of evictions from the managed region, *regardless of the number of partitions*! For example, if the cache has $R = 52$ candidates, with $A_{max} = 0.4$, we need to assign an extra $1/0.4 \cdot 52 = 4.8\%$ to the unmanaged region. Given that this is an acceptable size, we let partitions outgrow their allocations, disallowing inter-partition interference.

Transient behavior: So far, we have analyzed what happens in a *steady-state* situation. However, partitions may be suddenly *resized*. A partition that is suddenly downsized will need some time to reach its new target size (its aperture will be A_{max} during this period). Similarly, a partition that is suddenly upsized will take some time to acquire capacity (and will have an aperture of 0 until it reaches it). If we are reassigning space and upsized partitions gain capacity faster than downsized partitions lose it, the managed region may temporarily grow larger than it should be. In our evaluation, re-partitioning is infrequent and this is a minor issue. However, Vantage applications that resize partitions at high frequency should control the upsizing and downsizing of partitions progressively and in multiple steps.

Since partitions are cheap, some applications (e.g., local stores [42, 45]) might want to have a variable number of partitions, creating and deleting partitions dynamically. Deleting an existing partition simply requires setting its target size to 0, and its aperture to 1.0. When most or all of its lines have been demoted, the partition identifier can be reused for a new partition.

4.4 Vantage Cache Controller

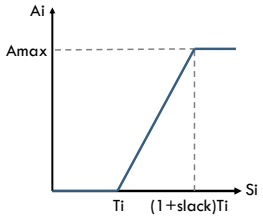
Vantage implements partitioning through the replacement process, so only the cache controller needs to be modified. Specifically, the controller is given the target sizes

of each partition and the partition ID of each cache access. Partition sizes are set by an external resource allocation policy (such as UCP), and partition IDs depend on the specific application. In our evaluation, we have one partition per thread, but other schemes may have other assignments, e.g., local stores [42, 45] may partition by address range, TM and TLS [32, 71] would have extra partitions to hold speculative data, etc. Vantage tags each line with its partition ID, and, on each replacement, performs evictions from the unmanaged region and demotions from the managed region, as described in Section 4.3. However, implementing a controller simply using the previous analysis is impractical due to several reasons:

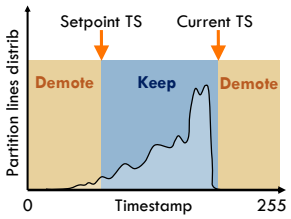
1. It is too compute-intensive: Each aperture A_i depends on the sizes and churns of all the other partitions (Equation 4.3 in Section 4.3.4), and they need to constantly change to adapt to time-varying behavior. Recomputing these on every replacement would be extremely expensive. Also, we need to estimate the churn (insertions/cycle) of each partition, which is not trivial.
2. It is not robust: The prior analysis has two sources of modeling errors. First, replacement candidates are not exactly independent and uniformly distributed (though they are close). Second, the previous analysis ignores promotions, which have no matching demotion[3]. Even if we could perfectly estimate the A_i, these modeling errors would cause partition sizes to drift away from their targets.
3. It requires knowing the eviction priority of every line (in order to know which candidates are below the aperture): This would be extremely expensive to do in practice.

In this section, we address these issues with a practical controller implementation that relies on two techniques: *feedback-based aperture control* enables a simple and robust controller where the required aperture is found using feedback instead of calculating it explicitly, and *setpoint-based demotions* lets us demote lines according to the desired aperture without knowing their eviction priorities.

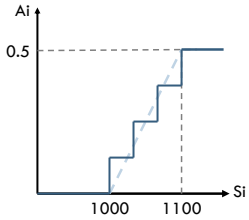
[3]One could argue that promotions are not bounded, so they may affect the strong guarantees derived in Section 4.3. Addressing this issue completely just requires to do one demotion per promotion on average, but we observe that in practice, promotions are rare compared to evictions, so demoting on evictions is enough for Vantage to work well.



(a) Linear transfer function used in feedback-based aperture control.



(b) Setpoint-based demotions selects candidates below setpoint (in modulo arithmetic).



Si range (lines)	Dems per 256 candidates
1000-1033	32
1034-1066	64
1067-1100	96
1101+	128

(c) 4-entry demotion thresholds lookup table for a 1000-line partition with 10% slack.

Figure 4.2: Feedback-based aperture control and setpoint-based demotions.

4.4.1 Feedback-based Aperture Control

Deriving the aperture of each partition is possible by using negative feedback alone. Once again, we let partitions slightly outgrow their target allocations, borrowing from the unmanaged region, and adjust their apertures based on how much they outgrow them. Specifically, we derive each aperture A_i as a function of S_i, as shown in Figure 4.2a:

$$A_i(S_i) = \begin{cases} 0 & \text{if} \quad S_i \leq T_i \\ \frac{A_{max}}{slack}\frac{S_i - T_i}{T_i} & \text{if} \quad T_i < S_i \leq (1+slack)T_i \\ A_{max} & \text{if} \quad S_i > (1+slack)T_i \end{cases} \tag{4.6}$$

where T_i is the partition's target size, and $slack$ is the fraction of the target size at which the aperture reaches A_{max} and tapers off. This is a classic application of negative feedback: an increase in size causes an increase in aperture, attenuating the size increase. The system is stable: partitions can reach and exceed a size of $(1+slack)T_i$, in which case A_{max} aperture is applied, and the dynamics of the system follow what was discussed in the previous section (i.e., the partition will reach a minimum stable size MSS_i). This linear transfer function is simple, works well in practice, and the extra space requirements are small and easily derived: in the linear

region, $\Delta S_i = S_i - T_i = slack \cdot S_i \frac{A_i}{A_{max}}$. Using Equation 4.3 (Section 4.3.4), we get:

$$\Delta S_i = \frac{slack}{A_{max}} S_i \frac{C_i \sum_{k=1}^{P} S_k}{S_i \sum_{k=1}^{P} C_k} \frac{1}{R \cdot m} = \frac{slack}{A_{max}} \frac{C_i}{\sum_{k=1}^{P} C_k} \frac{1}{R} \tag{4.7}$$

Therefore, the aggregate outgrow for all partitions in steady-state is:

$$\sum_{i=1}^{P} \Delta S_i = \frac{slack}{A_{max} \cdot R} \tag{4.8}$$

We will need to account for this when sizing the unmanaged region. This is relatively small, e.g., with $R = 52$ candidates, $slack = 0.1$ and $A_{max} = 0.4$, $\sum_{i=1}^{P} \Delta S_i = 0.48\%$ of the cache size. This also reveals the trade-off in selecting the slack: with a larger slack, apertures will deviate less from their desired value due to instantaneous size variations, but it requires a larger unmanaged region, as partitions will outgrow their target sizes by a larger amount. We will see how to size the unmanaged region in Section 4.4.3.

4.4.2 Setpoint-based Demotions

Setpoint-based demotions is a scheme to perform demotions without tracking eviction priorities. We first explain it with a concrete replacement policy, then generalize it to other policies.

We use coarse-timestamp LRU (Section 3.3.5) as the base replacement policy. Each partition has a *current timestamp* counter that is incremented every k_i accesses, and accessed lines are tagged with the current timestamp value. We choose 8-bit timestamps with $k_i = 1/16$ of the partition's size, which guarantees that wrap-arounds are rare. To perform demotions, we choose a *setpoint timestamp*, and all the candidates that are below it (in modulo 256 arithmetic) are demoted if the partition is exceeding its target size. We adjust the setpoint every c candidates seen from each partition in the following fashion: we have a counter for candidates seen from this partition, and a counter for the number of demoted candidates, d_i. Every time that the candidates counter reaches c, if $d_i > c \cdot A_i$ (i.e., $d_i/c > A_i$), the partition's setpoint

is incremented, and if $d_i < c \cdot A_i$, it is decremented. Both counters are then reset. Additionally, we increase the setpoint every time the timestamp is increased (i.e., every k_i accesses), so that the distance between both remains constant.

Figure 4.2b illustrates this scheme. Adjusting the setpoint allows us to track the aperture indirectly, without profiling the distribution of timestamps in the partition. In our controller, we find that $c = 256$ candidates is a sensible value. Since c is constant and, in our evaluation, target allocations are varied sparingly (every 5 million cycles), we do not even need to explicitly compute the desired aperture from the size (as in Equation 4.6). Instead, we use a small 8-entry *demotion thresholds lookup table* that gives the d_i threshold for different size ranges. Figure 4.2c shows a concrete example of this lookup table, where we have a partition with $T_i = 1000$ lines, and a 10% slack. For example, if when we reach $c = 256$ candidates from this partition, its size is anywhere between 1034 and 1066 lines, having more/less than 64 demotions in this interval will cause the setpoint to be incremented/decremented. This table is filled at resize time, and used every c candidates seen.

This scheme is also extensible to other policies beyond coarse-timestamp LRU. For example, in LFU we would choose a setpoint access frequency, and RRIP [85] can use a setpoint re-reference prediction value, as we will see in Section 4.6.

4.4.3 Putting it all Together

Now that we have seen the necessary techniques, we describe the implementation of the Vantage controller in detail.

State: Figure 4.3 shows the state required by Vantage:

- Tag state: Each line needs to be tagged with its partition ID, and we need an extra ID for the unmanaged region. For example, with $P = 32$ partitions, we need 33 identifiers, or 6 bits per tag. If tags are nominally 64 bits, and cache lines are 64 bytes, this is a 1.01% increase in cache state. Note that each tag also has an 8-bit timestamp field to implement the LRU replacement policy, as in the baseline zcache.
- Per-partition state: For each partition, the controller needs to keep track of the

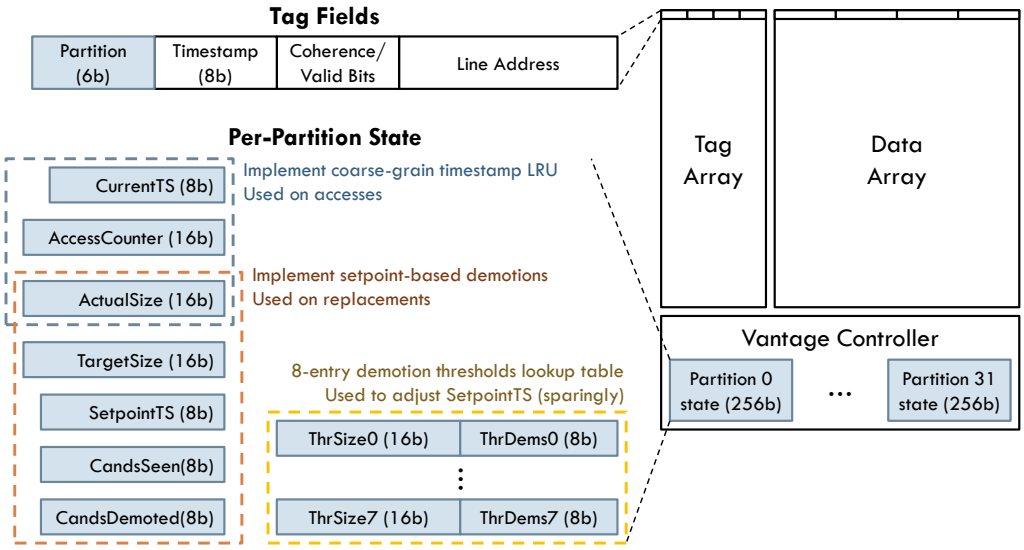


Figure 4.3: State required to implement Vantage: tag fields and per-partition registers. Additional state over an unpartitioned baseline is shown in blue. Each field or register shows its size in bits.

registers detailed in Figure 4.3. We explain how each of these registers is used below. Each register is labeled as either 8 or 16-bit, but 16-bit registers, which track sizes or quantities relative to size, assume a cache with 2^{16} lines. We assume that each of these registers is kept in partition-indexed register files. With $32K$ lines per bank, this amounts to 256 bits per partition. For 32 partitions and 4 banks (for an 8 MB cache), this represents 4 KBytes, less than a 0.5% state overhead.

Hits: On each hit, the controller writes the partition's *CurrentTS* into the tag's *Timestamp* field and increases the partition's *AccessCounter*. This counter is used to drive the timestamp registers forward: when *AccessCounter* reaches *ActualSize*/16, the counter is reset and both timestamp registers, *CurrentTS* and *SetpointTS*, are increased. This scheme is similar to the basic coarse-grained timestamp LRU replacement policy, except that the timestamp and access counter are per partition. Additionally, if the tag's *Partition* field indicates that the line was in the unmanaged region, this is a promotion, so *ActualSize* is increased and *Partition* is written when updating the *Timestamp* field.

Misses: On each miss, the controller examines the replacement candidates and performs one demotion on average, chooses the candidate to evict, and inserts the incoming line:

- All candidates are checked for demotion: a candidate from partition p is demoted when both *ActualSize[p]* > *TargetSize[p]* (i.e., the partition is over its target size) and the candidate's *Timestamp* field is not in between *SetpointTS[p]* and *CurrentTS[p]* (as shown in Figure 4.2b), which requires two comparisons to decide. If the candidate is demoted, the tag's *Partition* field is changed to the unmanaged region, its *Timestamp* field is updated to the unmanaged region's timestamp, *ActualSize[p]* is decreased, and *CandsDemoted[p]* is increased. Regardless of whether the candidate is demoted or not, *CandsSeen[p]* is increased.
- The controller evicts the candidate from the unmanaged region with the oldest timestamp. If all candidates come from the managed region, it chooses one of the demoted candidates arbitrarily, and if no lines are selected for demotion, it chooses among all the candidates. Note that if the unmanaged region is sized correctly, the common case is to find candidates from it.
- The incoming line is inserted into the cache as usual, with its *Timestamp* field set to its partition's *CurrentTS* register, and its *ActualSize* is increased. As in a hit, *AccessCounter* is increased and the timestamps are increased if it reaches *ActualSize*/16.

Additionally, to implement the setpoint adjustment scheme from Section 4.4.2, partition p's setpoint is adjusted when *CandsSeen[p]* crosses 0. At this point, the controller has seen 256 candidates from p since the last time it crossed 0 (since the counter is 8 bits), and has demoted *CandsDemoted[p]* of them. The controller finds the first entry K in the 8-entry demotion thresholds lookup table (as in Figure 4.2b) so that the partition's threshold size, *ThrSize[K][p]*, is lower than its current size, *ActualSize[p]*. It then compares *CandsDemoted[p]* with the demotion threshold, *ThrDems[K][p]*. If the demoted candidates exceed the threshold, *SetpointTS[p]* is decreased, while if they are below the threshold, the setpoint is increased. Finally, *CandsDemoted[p]* is reset. Note that this happens sparingly, e.g., if the cache examines 64 replacement candidates per miss, the controller does one setpoint adjustment each 256/64 = 4

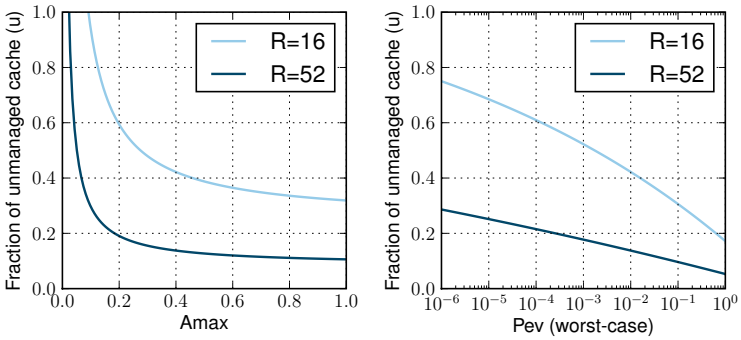


Figure 4.4: Fraction of the cache dedicated to the unmanaged region, with $slack = 0.1$ and $R = 16, 52$ candidates, both (a) as a function of A_{max}, with $P_{ev} = 10^{-2}$, and (b) as a function of P_{ev}, with $A_{max} = 0.4$.

misses on average, independently of the number of partitions.

Implementation costs: The controller requires counter updates and comparisons on either 8 or 16-bit registers, so a few narrow adders and comparators suffice to implement it. Operation on hits is simple and does not add to the critical path. On misses, demotion checks are the main overhead versus an unpartitioned cache, as the controller needs to decide whether to demote every candidate it sees, and each demotion check requires a few comparisons and counter updates. When a W-way zcache is used (typically $W = 4$ ways), replacements are done over multiple cycles, with the cache array returning at most W candidates per cycle. Therefore, a narrow pipeline suffices for demotions (i.e., we only need logic that can check $W = 4$ candidates per cycle). When using wider caches (e.g., a 16-way set-associative cache), the controller can implement demotion checks over multiple cycles, because the replacement process is not on the critical path. Finally, note that, while all candidates are checked for demotion, only one on average is demoted per miss. Unlike other partitioning schemes, Vantage does not need to implement set ordering or LRU chains or pseudo-random number generation [82, 128, 163].

Sizing the unmanaged region: We finally have all the information needed to size the unmanaged region. First, from Equation 3.1 (Section 4.3.2), to have a worst-case probability of a forced eviction from the managed region P_{ev}, we need $F_A(m) =$

Cores	32 cores, x86-64 ISA, in-order, IPC=1 except on memory accesses, 2 GHz
L1 caches	32 KB, 4-way set associative, split D/I, 1-cycle latency
L2 cache	8 MB NUCA , 4 banks, 2 MB per bank, shared, non-inclusive, MESI directory coherence, 4-cycle average L1-to-L2-bank latency, 8-cycle L2 bank latency
MCU	4 memory controllers, 200 cycles zero-load latency, 32 GB/s peak memory BW

Table 4.2: Main characteristics of the large-scale CMP. Latencies assume a 32 nm process at 2GHz.

$F_A(1-u) = P_{ev} = (1-u)^R$. Hence, at least we need $u \geq 1 - \sqrt[R]{P_{ev}}$. Additionally, we need to reserve $1/(A_{max}R)$ to allow high-churn/small-sized partitions to grow to their minimum stable sizes, and $slack/(A_{max}R)$ for feedback-based aperture control. Sizing $u = 1 - \sqrt[R]{P_{ev}} + (1+slack)/(A_{max}R)$ accounts for all these effects. Figure 4.4 shows the fraction of the cache that needs to be unmanaged when varying both A_{max} and P_{ev}, for a 10% slack and $R = 16$ or 52 candidates. For example, with 52 candidates, having $A_{max} = 0.4$ requires 13% of the cache to be unmanaged for $P_{ev} = 10^{-2}$, while going down to $P_{ev} = 10^{-4}$ would require 21% to be unmanaged. Different applications will have different requirements for P_{ev}. For example, $P_{ev} \simeq 10^{-2}$ may suffice for applications that only require approximate partitioning, while applications with strong partitioning and isolation requirements may need $P_{ev} \simeq 10^{-4}$ or lower.

4.5 Experimental Methodology

Modeled systems: We model both small and large-scale CMPs using zsim (Section 3.5). Our large-scale design has 32 in-order, single-threaded x86 cores modeled after Atom [62]. The system has private L1s and a shared 8MB, 4-bank L2, where the different partitioning schemes are implemented. Table 4.2 shows the details of the system. On a high-performance 32nm process, this CMP requires about 220 mm^2 and has a TDP of around 90W at 2GHz. Our small-scale design is similar, but has 4 cores, a 2MB L2 (1 bank) and 4GB/s of memory bandwidth.

Partitioning schemes: We compare Vantage against way-partitioning and PIPP. Way-partitioning uses LRU, and its replacement process is implemented as in [128]. PIPP is implemented as described in [163] ($p_{prom} = 3/4$, stream detection with $\theta_m \geq$

Insensitive (n)	perlbench, bwaves, gamess, gromacs, namd, gobmk, dealII, povray, calculix, hmmer, sjeng, h264ref, tonto, wrf
Cache-friendly (f)	bzip2, gcc, zeusmp, cactusADM, leslie3d, astar
Cache-fitting (t)	soplex, lbm, omnetpp, sphinx3, xalancbmk
Thrashing/streaming (s)	mcf, milc, GemsFDTD, libquantum

Table 4.3: Classification of SPEC CPU2006 workloads.

12.5%, 1 way per streaming application and $p_{stream} = 1/128$).

Allocation policy: We use utility-based cache partitioning (UCP) to determine space allocation among partitions [128]. UCP uses auxiliary cache monitors to estimate how well each core uses cache capacity, and allocates more capacity to the threads that benefit from it the most. Each core has a small utility monitor based on dynamic set sampling (UMON-DSS) with 64 sets. Partition sizes are found with the Lookahead algorithm [128]. UCP repartitions the cache every 5 million cycles. When used with Vantage, UMONs are configured with the same number of ways as way-partitioning and PIPP are using, but since Vantage can partition at line granularity instead of at way granularity, we linearly interpolate the miss rate curves given by UMON, getting 256-point curves, and use them to drive the Lookahead algorithm.

Workloads: We use multiprogrammed SPEC CPU2006 application mixes, and follow the methodology of prior cache partitioning studies [128, 163]. Each application in the mix is fast-forwarded for 20 billion instructions, and the mix is simulated until all applications have executed 200 million instructions. We report aggregate throughput ($\sum IPC_i$), where each application's IPC is measured on its first 200 million instructions. Other studies also report metrics that give insight on fairness, such as weighted speedup or harmonic mean of weighted speedups [128, 163]. Due to lack of space, and because UCP attempts to maximize throughput, we report throughput only. We have checked these metrics and they do not offer additional insights. Fairness is mostly an issue of the *allocation policy*, i.e., UCP.

The 29 SPEC programs are divided in four categories, following a classification similar to the one in [83]. We first run each application alone, using cache sizes from 64KB to 8MB. Applications with less than 5 L2 misses per kilo-instruction (MPKI)

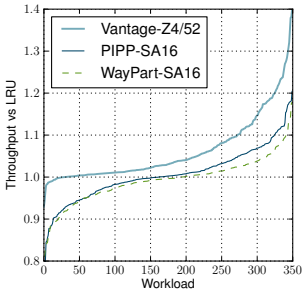
are classified as *insensitive*; from the remaining ones, applications that gradually benefit from increased cache size are classified as *cache-friendly*; those where misses decrease abruptly with size when getting close to cache capacity (over 1MB) are classified as *cache-fitting*, and the ones where additional capacity does not yield any benefit are marked as *thrashing/streaming*. Table 4.3 shows this classification. There are 35 possible combinations (with repetitions) of these four categories, each of which forms a class. In the 4-core mixes, we have 10 mixes per class, with each application being randomly selected from the ones in its category, yielding 350 workloads. The 32-core mixes have 8 randomly chosen workloads per category, and again 10 mixes per class, for another 350 workloads.

4.6 Evaluation

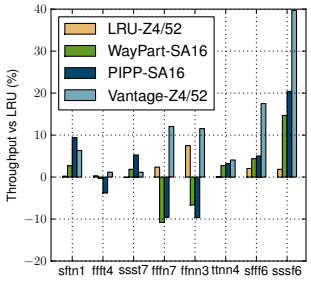
We first compare Vantage against other partitioning schemes using utility-based cache partitioning. We then present a series of experiments focused on Vantage, showing how to configure it, its sensitivity to configuration parameters, and confirm that the assumptions made in the analysis are met in practice.

4.6.1 Comparison of Partitioning Schemes

Small-scale configuration: Figure 4.5a summarizes the performance results across the 350 workload mixes on the simulated 4-core system. Each line shows the throughput ($\sum IPC_i$) of a different scheme, normalized to a 16-way set-associative cache using LRU. For each line, workloads (the x-axis) are sorted according to the improvement achieved. All caches use simple H_3 hashing [31], since it improves performance in most cases. Way-partitioning and PIPP use a 16-way set-associative cache, while Vantage uses a 4-way zcache with 52 replacement candidates (Z4/52), with a $u = 5\%$ unmanaged region, $A_{max} = 0.5$ and $slack = 10\%$. Although zcaches have a lower hit latency, we simulate the same hit latency for all designs (which is unfair to Vantage, but lets us isolate the improvements due to partitioning).



(a) Results over all 350 workloads



(b) Throughput for selected workloads

Figure 4.5: Throughput improvements over an unpartitioned 16-way set-associative L2 with LRU obtained with different partitioning schemes on the 4-core configuration.

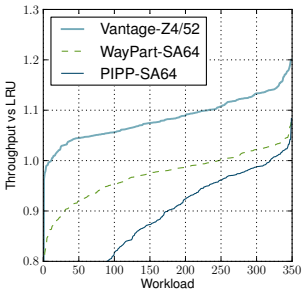


Figure 4.6: Throughput improvements over a 64-way set-associative L2 with LRU on the 32-core configuration.

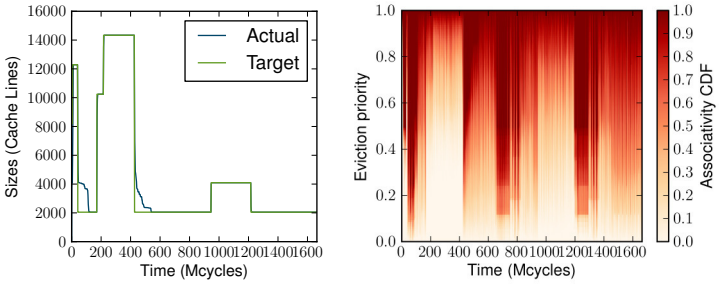
Figure 4.5a shows that, overall, Vantage provides much larger improvements than either way-partitioning or PIPP: a 6.2% geometric mean on average and up to 40%. While Vantage slightly decreases performance for only 4% of the workloads, when using either way-partitioning or PIPP, around 45% of the workloads show worse throughput, often significantly (up to 22% worse for way-partitioning, and 29% worse for PIPP). These workloads already share the cache efficiently with LRU, and partitioning hurts performance by decreasing associativity. Indeed, when using 64-way set-associative caches, way-partitioning and PIPP improve performance for most workloads. This shows the importance of maintaining high associativity, which Vantage achieves.

Figure 4.5b compares the throughput of selected workload mixes. Each bar represents throughput improvements of a specific configuration, and there is an additional configuration per set, an unpartitioned Z4/52 zcache, to determine how much the higher associativity of the zcache is helping Vantage. As we can see, most of the benefits are due to Vantage, not the zcache, though they are complementary. We have selected these workloads to illustrate several points. First, we observe that PIPP sometimes shows significantly different behavior from way-partitioning and Vantage, sometimes outperforming both (`sftn1`), and sometimes doing considerably

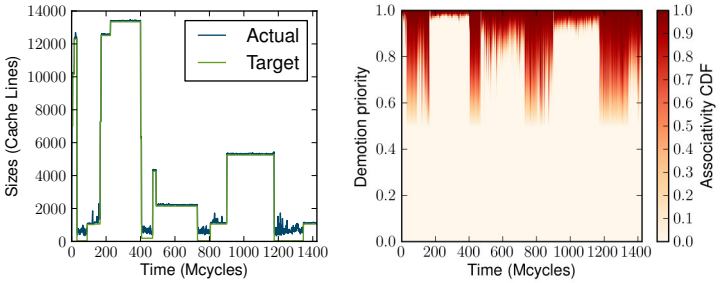
worse (`ffft4`). PIPP does not use LRU, and performance differences do not necessarily come from partitioning. Nevertheless, both way-partitioning and Vantage can benefit from another replacement policy, as we will see in Section 4.6.2. Between way-partitioning and Vantage, Vantage achieves higher performance in all except 3 of the 350 workloads. In these rare cases (e.g., `ssst7`), way-partitioning has a slight edge as Vantage cannot partition the whole cache, which affects some mixes, especially those with cache-fitting applications where the miss rate curve decreases abruptly. Way-partitioning and PIPP, however, do significantly worse on associativity-sensitive workloads, such as `fffn7` and `ffnn3`. We can see that, in these cases, the highly-associative zcache has a more noticeable effect in improving Vantage's performance. Finally, mixes `ttnn4`, `sfff6` and `sssf6` illustrate typical behavior of workloads that benefit more from partitioning than from high associativity: both way-partitioning and PIPP improve performance, with PIPP having a slight edge over way-partitioning, while Vantage provides significantly higher throughput.

Large-scale configuration: Figure 4.6 shows the throughput improvements of different partitioning schemes for the 32-core system, in the same fashion as Figure 4.5a. In this configuration, the baseline, way-partitioning and PIPP configurations use a 64-way cache, while Vantage uses *the same* Z4/52 zcache and configuration of the 4-core experiments. Results showcase the scalability of Vantage: while way-partitioning and PIPP degrade performance for most workloads, even with their highly-associative caches, Vantage continues to provide significant improvements on most workloads (8.0% geometric mean and up to 20%) with the same configuration as the 4-core system. While low associativity is again the culprit with way-partitioning, PIPP has much more severe slowdowns (up to 3×) because its approach of assigning an insertion position equal to the number of allocated ways causes very low insertion positions with many partitions, leading to high contention at the lower end of the LRU chain and hard to evict dead lines at the higher end.

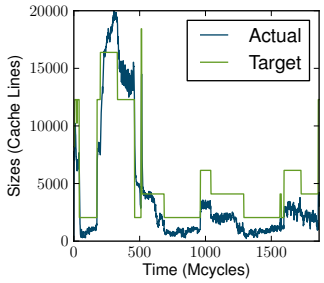
Partition sizes and associativity: Figure 4.7 shows, for each partitioning scheme, the target and actual partition sizes as a function of time for a specific partition and workload mix in the 4-core system. As we can see, way-partitioning and Vantage



(a) Way-partitioning



(b) Vantage



(c) PIPP

Figure 4.7: Comparison of way-partitioning, Vantage and PIPP for a specific partition in a 4-core mix. Plots show target partition size (as set by UCP) and actual size for the three schemes. We also show heat maps of the measured associativity CDF on this partition for way-partitioning and Vantage.

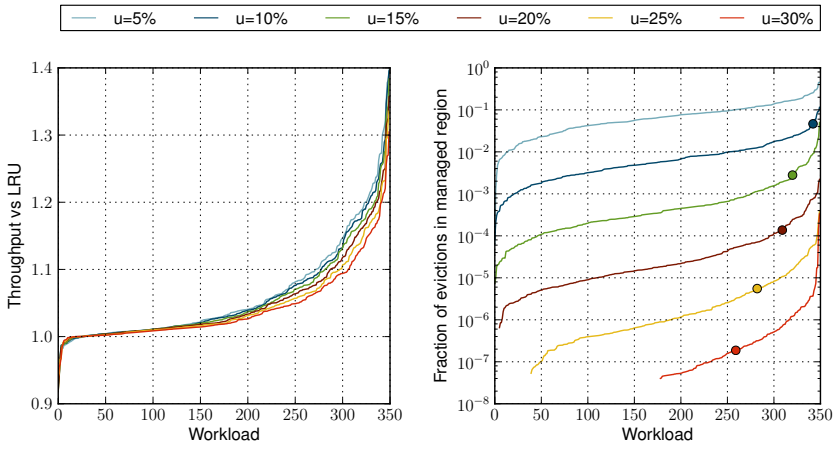
closely track the target size, while PIPP only approximates it. More importantly, in Vantage the partition is never under its target allocation, while in PIPP the target is often not met (e.g., in some intervals the target size is 2048 lines, but the partition has less than 100). We also observe that with way-partitioning, when the target size is suddenly decreased, reaching the new target allocation can take a significant

amount of time (100 Mcycles). This happens because the applications that now own the reallocated ways need to access all the sets and evict all of this partition's lines in those ways. In contrast, Vantage adapts much more quickly, both because of the better location randomization of zcaches and because it works on global, not per-set, allocations. Finally, at times UCP gives a negligible allocation to this partition (128 lines in Vantage, 2048 lines, i.e., 1 way in way-partitioning/PIPP). Vantage cannot keep the partition size that small, so it grows to its minimum stable size, which hovers around 400-700 lines. In this cache, the worst-case minimum stable size is $1/(A_{max}R) = 1/0.5 \cdot 52 = 3.8\%$, i.e., 1260 lines, but replacements caused by other partitions help this partition stay smaller.

Figure 4.7 also shows the time-varying behavior of the associativity distributions on way-partitioning and Vantage using heat maps. For each million cycles, we plot the portion of eviction/demotions that happen to lines below a given eviction/demotion priority (i.e., the empirical associativity CDFs). For a given point in time (x-axis), the higher in the y-axis the heat map starts becoming darker, the more skewed the demotion/eviction priorities are towards 1.0, and the higher the associativity. Vantage achieves much higher associativity than way-partitioning: when the partition is large (7 ways at 200-400 Mcycles), way-partitioning gets acceptable associativity, but when given one way, evictions have almost uniformly distributed eviction priorities in $[0, 1]$, and even worse at times (e.g., 700-800 Mcycles). In contrast, Vantage maintains a very high associativity when given a large allocation (at 200-400 Mcycles, the aperture hovers around 3%) because the churn/size ratio is low. Even when given a minimal allocation, demoted lines are uniformly distributed in $[0.5, 1]$, by virtue of the maximum aperture, giving acceptable worst-case associativity.

4.6.2 Vantage Evaluation

Sensitivity analysis: Figure 4.8a shows the performance of Vantage on the 4-core workloads when the size of the unmanaged region changes from 5% to 30% in a Z4/52 zcache. Differences are relatively small, and a size of 5% delivers the highest throughput. Figure 4.8b shows what portion of evictions happen from the managed



(a) Throughput improvement (b) Evictions in managed region

Figure 4.8: Throughput and fraction of evictions in the managed region when varying the size of the unmanaged region, on a Z4/52 cache with $A_{max} = 0.5$ and $slack = 0.1$.

region (because no candidates are from the unmanaged region). For $u = 5\%$, on most workloads 1% to 10% of the evictions come from the managed region. By having a smaller unmanaged region, Vantage can partition a larger portion of the cache, but this slightly degrades isolation. UCP is not very sensitive to strict isolation or partition size control, but benefits from having more space to partition, so 5% works best. Other applications may need better isolation, which would require a larger unmanaged region. We have also studied the sensitivity of Vantage to the maximum aperture, A_{max}, and the *slack* needed for feedback-based aperture control. With UCP, Vantage is largely insensitive to these parameters: ranges of $5 - 70\%$ for A_{max} and $slack > 2\%$ work well.

Comparison with analytical models: In Figure 4.8b, we have included a round marker at the point where each line crosses the worst-case eviction priority P_{ev}, as predicted by our models (Section 4.4.3). Most workloads achieve probabilities below the predicted worst-case. For those that exceed it, we have determined that frequent transients are the main culprit: these workloads have fast time-varying behavior, UCP continuously changes target sizes, and the size of the unmanaged region shrinks during transients, increasing evictions. Nevertheless, Figure 4.8b shows that we can

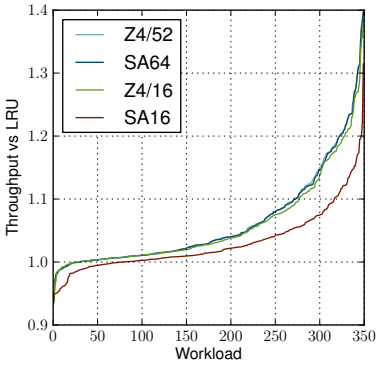


Figure 4.9: Throughput improvements of Vantage on the 4-core system, using different cache designs.

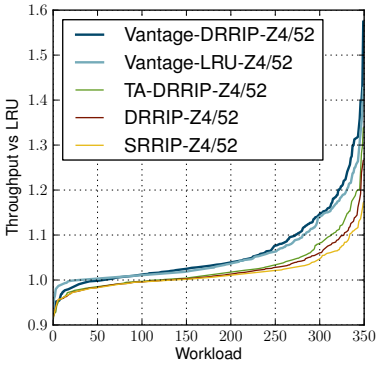


Figure 4.10: Throughput improvements on the 4-core system using RRIP variants and Vantage.

make evictions in the managed region arbitrarily rare by increasing the size of the unmanaged region, achieving strong isolation guarantees.

We also simulated Vantage in two unrealistic configurations to test that our assumptions hold: first, using feedback-based aperture control but with perfect knowledge of the apertures instead of using setpoint-based demotions, and second, using a *random candidates* cache, an unrealistic cache design that gives truly independent and uniformly distributed candidates. Both design points perform exactly as the practical implementation of Vantage. These results show that our simple controller provides the benefits predicted by the analytical models.

Set-associative and low-associativity caches: Figure 4.9 compares Vantage on different cache designs on the 4-core system: our original Z4/52 zcache; a Z4/16 zcache, and 64 and 16-way set-associative caches. Vantage is tuned in each case: the 16-way set-associative and Z4/16 caches use an unmanaged region $u = 10\%$, while the 64-way set-associative and Z4/52 caches use $u = 5\%$. All use $A_{max} = 0.5$ and $slack = 0.1$. As we can see, Vantage works well on set-associative caches and degrades gracefully with lower associativity: the 64-way set-associative cache and Z4/52 zcache achieve practically the same performance, followed very closely by the Z4/16 design, and the 16-way set-associative does sensibly worse, although still significantly better than either way-partitioning or PIPP with a 16-way cache (Figure 4.5a). These results

show that, although Vantage works best and provides stronger isolation with zcaches, it is practical to use with traditional set-associative caches.

Comparison with alternative replacement policies: We have used LRU so far because the partitioning schemes we compare Vantage with are based on LRU. However, much prior work has improved on LRU, both in performance and implementation cost. The RRIP family of replacement policies [85] is one such example. They include scan-resistant SRRIP, thrash-resistant BRRIP, and scan and thrash-resistant DRRIP, which uses set dueling to choose between SRRIP and BRRIP dynamically. Additionally, TA-DRRIP enhances performance in shared caches by using TADIP's thread-aware set dueling mechanism on DRRIP [83]. These policies do not require set ordering, so they are trivially applicable to zcaches and Vantage. Figure 4.10 compares the performance achieved by using these policies with two variants of Vantage, one using LRU and other using DRRIP. All RRIP variants use a 3-bit re-reference prediction value (RRPV) in each tag instead of 8-bit LRU timestamps. In Vantage-DRRIP, we have a per-partition setpoint RRPV instead of a setpoint LRU timestamp, and do not age lines from partitions below their target size, but otherwise the scheme works as in [85]. Additionally, for Vantage-DRRIP to work, we have to (1) modify UCP's UMON-DSS mechanism to work with RRIP instead of LRU, and (2) provide a way to decide between SRRIP and BRRIP. To achieve this, UMON-DSS is modified to maintain RRIP chains instead of LRU chains (i.e., lines are ordered by their RRPVs), and one half of the UMON sets use SRRIP, while the other half use BRRIP. Each time partitions are resized, the best of the two policies is chosen for each partition and used in the next interval. Because the decision of whether to use SRRIP or BRRIP is done per partition, Vantage-DRRIP is automatically thread-aware.

Figure 4.10 shows that Vantage-LRU outperforms all RRIP variants, and Vantage-DRRIP further outperforms Vantage-LRU, although by a small amount: the geometric means over all benchmarks are 2.5% for TA-DRRIP, 6.2% for Vantage-LRU, and 6.8% for Vantage-DRRIP. We can extract three conclusions from these experiments. First, Vantage can be easily modified to work with alternative replacement policies. Second, Vantage is still beneficial when using a better replacement policy. Moreover,

partitioning has several other uses beyond improving miss rates, as explained in Section 4.1. Finally, we note that these results are preliminary, as there may be better ways than using UMON to decide partition sizes and choosing the replacement policy.

4.7 Summary

We have presented Vantage, a scalable and efficient scheme for fine-grained cache partitioning. Vantage works by matching the insertion (churn) and demotion rates of each partition, thus keeping their sizes approximately constant. It partitions most of the cache, and uses the unmanaged region to eliminate inter-partition interference and achieve a simple implementation. Vantage is derived from analytical models, which allow it to provide different degrees of isolation by varying the size of the unmanaged region: a small unmanaged region (5%) suffices to provide moderate isolation, while a larger region (20%) can provide strong isolation and eliminate inter-partition interference. Thus, Vantage satisfies the needs of applications with different isolation requirements, all while maintaining a good associativity per partition regardless of the number of partitions. Under UCP, Vantage outperforms existing partitioning schemes on small-scale CMPs, but most importantly, it continues to deliver the same benefits on CMPs with tens of threads, where previous schemes fail to scale.

Chapter 5

SCD: Scalable Coherence Directory with Flexible Sharer Set Encoding

5.1 Introduction

Implementing coherent cache hierachies becomes increasingly difficult as we scale CMPs into the hundreds and thousands of cores. Large-scale CMPs require a directory-based protocol, which introduces a coherence directory between the private and shared cache levels to track and control which caches share a line and serve as an ordering point for concurrent requests. However, as we discussed in Section 2.2.3 implementing directories that can track hundreds of sharers efficiently has been problematic so far. Moreover, directories are *fully shared* resources, so, like shared caches, they can introduce a significant amount of interference across competing applications in the form of invalidations in the lower levels of the cache hierarchy. Previously proposed directory implementations do not address this issue, precluding a cache coherent CMP that provides QoS guarantees.

In this chapter, we present the Scalable Coherence Directory (SCD), a novel directory scheme that scales to thousands of cores efficiently, while incurring negligible invalidations and keeping an exact sharer representation. We leverage zcache's analytical properties to design and analyze SCD, and show that it can provide performance guarantees with a minimal amount of overprovisioning. This chapter presents the

following contributions:

1. We present the SCD design. We recognize that, to be scalable, a directory implementation only needs the number of bits *per tracked sharer* to scale gracefully (e.g., remaining constant or increasing logarithmically) with the number of cores. SCD exploits this insight by using a *variable-size sharer set representation*: lines with one or few sharers use a single directory tag, while widely shared lines use additional tags. We propose a hybrid pointer/bit-vector organization that scales logarithmically and can track tens of thousands of cores efficiently. While conventional set-associative arrays have difficulties with this approach, highly-associative zcache arrays allow SCD to work.
2. We develop analytical models that characterize SCD and show how to size it. First, we show that for a given occupancy (fraction of directory capacity used), SCD incurs the same number of directory-induced invalidations and average number of lookups, independently of the workload. Second, different workloads impose varying capacity requirements, but the worst-case capacity requirement is bounded and small. Hence, directories can be built to guarantee a negligible number of invalidations and a small average number of lookups in all cases, guaranteeing performance and energy efficiency with just a small amount of overprovisioning (around 5-10% depending on the requirements, much smaller than what is required with set-associative arrays [59]). These results are useful for two reasons. First, they enable designers to quickly size directories without relying on empirical results and extensive simulations. Second, they *provide guarantees on the interference introduced by the shared directory*, which is paramount to achieve performance isolation among multiple competing applications sharing the chip (CMPs need a fully shared directory even if they have private last-level caches). This analytical characterization also applies to sparse directories implemented with zcaches.
3. We evaluate SCD by simulating CMPs with 1024 cores and a 3-level cache hierarchy. We show that, for the same level of provisioning, SCD is 13× more area-efficient than sparse directories and 2× more area-efficient than hierarchical

Scheme	Scalable area	Scalable energy	Exact sharers	Dir-induced invals	Extra protocol complexity	Extra latency
Duplicate-tag	Yes	No	Yes	No	No	No
Sparse full-map	No	Yes	Yes	Yes	No	No
Coarse-grain/limptr	No	Yes	No	Yes	Small	No
Hierarchical sparse	Yes	Yes	Yes	Yes	High	Yes
Tagless	No	No	No	Yes	Medium	No
SPACE	No	Yes	No	Yes	Small	No
Cuckoo Directory	No	Yes	Yes	Negligible	No	No
SCD	Yes	Yes	Yes	Negligible	No	No

Table 5.1: Qualitative comparison of directory schemes. The first three properties are desirable, while the last three are undesirable.

organizations. SCD can track 128MB of private cache space with a 20MB directory, taking only 3% of total die area. Moreover, we show that the analytical models on invalidations and energy are accurate in practice, enabling designers to guarantee bounds on performance, performance isolation, and energy efficiency.

5.2 Background on Directory Organizations

Cache coherence is needed to maintain the illusion of a single shared memory on a system with multiple private caches. A coherence protocol arbitrates communication between the private caches and the next level in the memory hierarchy, typically a shared cache (e.g., in a CMP with per-core L2s and a shared last-level cache) or main memory (e.g., in multi-socket systems with per-die private last-level caches). In this work we focus on *directory-based*, *write-invalidate* protocols, as alternative protocols scale poorly beyond a few private caches [73]. These protocols use a *coherence directory* to track which caches share a line, enforce write serialization by invalidating or downgrading access permissions for sharers, and act as an ordering point for concurrent requests to the same address. Implementing a directory structure that scales to hundreds of sharers efficiently has been problematic so far. We now review different directory organizations, with a focus on comparing their scalability. Table 5.1 summarizes their characteristics.

Traditional directory schemes do not scale well with core count. *Duplicate-tag* directories maintain a full copy of all the tags tracked in the lower level. Their area requirements scale well with core count, but they have huge associativity requirements (e.g., tracking 1024 16-way caches would require 16384 ways), so they are limited to small-scale systems [13, 149]. In contrast, *sparse directories* [69] are organized as an associative array indexed by line address, and each directory tag encodes the set of sharers of a specific address. Sparse directories are energy-efficient. However, due to their limited associativity, sparse directories are often forced to evict entries, causing *directory-induced invalidations* in the lower levels of the hierarchy. This can pose large performance overheads and avoiding it requires directories that are significantly overprovisioned [59].

The encoding method for the sharer set is a fundamental design choice in sparse directories. *Full-map* sparse directories encode the sharer set exactly in a bit-vector [69]. They support all sharing patterns, but require storage proportional to the number of cores, and scale poorly beyond a few tens of cores. Alternatively, sparse directories can use a compressed but inexact encoding of the sharer set. Traditional alternatives include *coarse-grain sharer bit-vectors* [69], and *limited pointer* schemes, in which each entry can hold a small number of sharer pointers, and lines requiring more sharers either cause one of the existing sharers to be invalidated, a broadcast on future invalidations and downgrades [3, 102], or trigger an interrupt and are handled by software [33]. Inexact sharer set schemes trade off space efficiency for additional coherence traffic and protocol complexity.

In contrast to these techniques, *hierarchical sparse directories* [66, 158, 164] allow an exact and area-efficient representation of sharer sets. Hierarchical directories are organized in multiple levels: each first-level directory encodes the sharers of a subset of caches, and each successive level tracks directories of the previous level. A two-level organization can scale to thousands of cores efficiently. For example, using 32 first-level directories and one (possibly banked) second-level directory, we can track 1024 caches using 32-bit sharer vectors, or 4096 caches using 64-bit vectors. However, hierarchical directories have two major drawbacks. First, they require several lookups on the critical path, increasing directory latency and hurting performance. Second,

multi-level coherence protocols are more complex than single-level protocols, and significantly harder to verify [40, 41].

Motivated by the shortcomings of traditional approaches, recent work has investigated alternative directory organizations that improve scalability in a single-level directory. Tagless directories [170] use Bloom filters to represent sharer sets. Tagless does not store address tags, making it highly area-efficient (as we will see later, SCD spends more space storing addresses than actual coherence information). Although Tagless reduces area overheads, both area and energy scale linearly with core count, so Tagless is area-efficient but not energy-efficient at 1024 cores [59]. Additionally, it requires significant modifications to the coherence protocol, and incurs additional bandwidth overheads due to false positives. Moreover, Tagless relies on the tracked caches being set-associative, and would not work with other array designs, such as zcaches. SPACE [172] observes that applications typically exhibit a limited number of sharing patterns, and introduces a level of indirection to reduce sharing pattern storage: a sharing pattern table encodes a limited number of sharing patterns with full bit-vectors, and an address-indexed sparse directory holds pointers to the pattern table. Due to the limited sharing pattern table size, patterns often need to be merged, and are inexact. However, multiple copies of the sharing pattern table must be maintained in a tiled organization, increasing overheads with the number of tiles [172]. Although these schemes increase the range of sharers that can be tracked efficiently, they are still not scalable and require additional bandwidth.

Alternatively, prior work has proposed coarse-grain coherence tracking [29, 54, 169]. These schemes reduce area overheads, but again increase the number of spurious invalidation and downgrade messages, requiring additional bandwidth and energy. Finally, to reduce directory overheads, WayPoint [91] proposes to cache a sparse full-map directory on the last-level cache, using a hash table organization. This reduces directory overheads and works well if programs have significant locality, but it reduces directory coverage and introduces significant complexity.

Finally, Cuckoo Directory [59] uses skew-associative arrays with W-ary Cuckoo hashing instead of set-associative arrays to build sparse directories. As we saw in

Section 3.3.4, zcache is much better suited for a hardware implementation than conventional Cuckoo hashing. ZCache requires fewer lookups and far fewer moves (saving energy), can be pipelined (reducing latency), and enables using a replacement policy. For example, in a 4-way array, evaluating 52 replacement candidates requires 13 lookups and at most 2 moves in a zcache, but 17 lookups and 16 moves in a Cuckoo Directory. However, the Cuckoo Directory replacement process can stop early, while zcaches expand a fixed number of candidates. SCD combines the best features from both schemes to implement its replacement process. Ferdman et al. empirically show that Cuckoo Directory reduces evictions (and therefore directory-induced invalidations) to negligible levels with arrays that are somewhat larger than the caches they are tracking. SCD's analytical models are also applicable to Cuckoo Directories, providing an analytical foundation and enabling tighter sizing.

As Table 5.1 shows, all these schemes suffer from one or several significant drawbacks. In contrast, SCD, which we present in this paper, represents sharer sets exactly and in a scalable fashion (both in area and energy), does not require coherence protocol modifications, and can be designed to guarantee an arbitrarily small amount of invalidations. SCD's design relies on the flexibility provided by efficient highly-associative caches, which we review next.

5.3 Scalable Coherence Directory

To scale gracefully, SCD exploits the insight that a directory does not need to provide enough capacity to track a specific number of addresses, but a specific number of *sharers*, ideally as many as can fit in the tracked caches. However, sparse directories use address-indexed arrays, so they use one directory tag per address. Instead, SCD represents sharer sets using a variable number of tags per address. Lines with one or a few sharers use a single directory tag with a limited pointer format, and widely shared lines employ a multi-tag format using hierarchical bit-vectors. We first describe the array used to hold the directory tags, then explain how SCD represents and operates on sharer sets.

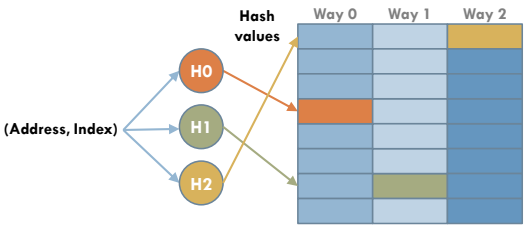


Figure 5.1: Example 3-way array organization used.

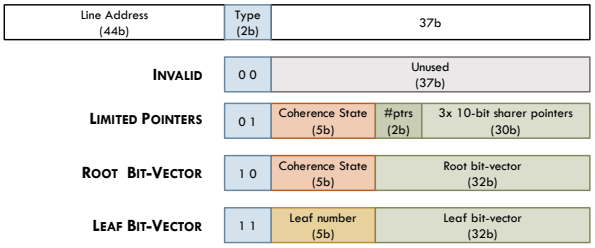


Figure 5.2: SCD line formats. Field widths assume a 1024-sharer directory.

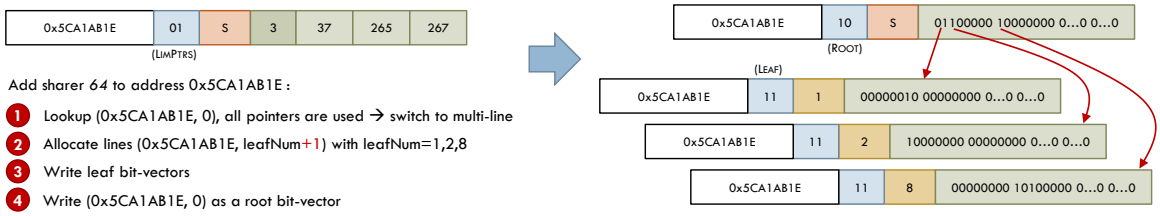


Figure 5.3: Example operation: adding a sharer to a full limited pointer line.

5.3.1 SCD Array

Figure 5.1 shows the structure of the SCD array. It is similar to a zcache, with one hash function per way. However, hash functions take the concatenation of the line address and an *index* as input instead of using just the address. Every line in the directory will have a tag with index 0. Additionally, lines that use more than one directory tag will have those additional tags at locations with indices other than 0. These indices need not be consecutive, and are included in the hash functions so that multiple tags representing the same line map to different sets.

SCD's replacement process is very similar to zcache's, since it is more efficient than Cuckoo Directory's, as explained in Section 5.2. However, SCD does not pipeline the replacement process, and stops looking for candidates as soon as an empty tag is found. As we will see, this greatly improves directory efficiency. SCD can optionally implement a replacement policy. However, replacement policies only make sense for underprovisioned directories — in Section 5.4 we will see that SCD can be sized to cause a negligible amount of evictions regardless of the workload, making a replacement policy unnecessary.

5.3.2 Line Formats

SCD encodes lines using different tag formats. Figure 5.2 illustrates these formats for a 1024-sharer directory. Lines with few sharers use a single-tag, *limited pointer* representation, with three pointers in the example. When more sharers are needed, SCD switches to a multi-tag format using hierarchical bit-vectors. In the example, a 32-bit *root* bit-vector tag indicates which subsets of cores share the line, while a set of 32-bit *leaf* bit-vectors encode the sharers in each subset. Leaf bit-vectors include a *leaf number* field that encodes the subset of sharers tracked by each leaf (e.g., leaf number 0 tracks sharers 0–31, 1 tracks 32–63, and so on). We will explain SCD's operation using this two-level representation, but this can be easily extended to additional levels.

5.3.3 Directory Operations

SCD needs to support three operations on sharer sets: adding a sharer when it requests the line, removing a sharer when it writes back the line, and retrieving all sharers for an invalidation or downgrade.

Adding and removing sharers: On a directory miss, the replacement process allocates one tag for the incoming line with index 0 (possibly evicting another tag). This tag uses the limited pointer format. Further sharers will use additional pointers. When a sharer needs to be added and all the pointers are used, the line is switched to the multi-tag format as follows: First, the necessary bit-vector leaves are allocated for the existing pointers and the new sharer. Leaf tags are then populated with the existing and new sharers. Finally, the limited pointer tag transitions to the root bit-vector format, setting the appropriate bits to 1. Figure 5.3 illustrates this process. Removing a sharer (due to clean or dirty writebacks) follows the inverse procedure. When a line loses all its sharers, all its directory tags are marked as invalid.

Invalidations and downgrades: Invalidations are caused by both coherence (on a request for exclusive access, the directory needs to invalidate all other copies of the line) and evictions in the directory. Downgrades only happen due to coherence

(a request for read on an exclusive line needs to downgrade the exclusive sharer, if any). Coherence-induced invalidations are trivial: all the sharers are sent invalidation messages. If the address is represented in the hierarchical bit-vector format, all leaf bit-vectors are marked as invalid, and the root bit-vector tag transitions to the limited pointer format, which then encodes the index of the requesting core.

In contrast, eviction-induced invalidations happen to a specific *tag*, not an address. Limited pointer and root bit-vector tag evictions are treated like coherence-based invalidations, invalidating all sharers so that the tag can be reused. Leaf bit-vector evictions, however, only invalidate the subset of sharers represented in the tag. As we will see later, eviction-induced invalidations can be made arbitrarily rare.

Additional levels and scalability: SCD can use hierarchical bit-vector representations with more than two levels. A two-level approach scales to 256 sharers with ∼16 bits devoted to track sharers (pointers/bit-vectors) per tag, 1024 sharers with ∼32 bits, and 4096 sharers with ∼64 bits. A three-level representation covers 4096 sharers with ∼16 bits, 32768 sharers with ∼32 bits, and 256K sharers with ∼64 bits. Four-level implementations can reach into the millions of cores. In general, space and energy requirements increase with $O(logN)$, where N is the number of sharers, because the limited bit-vectors and the extended address space increase logarithmically. Since each tag needs on the order of 40-50 bits to store the line address anyway, having on the order of 16-32 bits of sharer information per tag is a reasonable overhead.

5.3.4 Implementation Details

SCD's multi-tag format achieves the scalability of hierarchical directories, but since *all tags are stored in the same array, it can be made transparent to the coherence protocol*. We now discuss how to implement SCD to make it completely transparent to the protocol, and take the delay of additional accesses out of the critical path, providing the performance of a sparse directory.

Scheduling: Directories must implement some scheduling logic to make operations appear atomic to the protocol while ensuring forward progress and fairness. This

is required in both sparse directories and SCD. For example, in a sparse directory adding a sharer is a read-modify-write operation, and the scheduling logic must prevent any intervening accesses to the tag between the read and the write (e.g., due to a conflicting request or an eviction). However, because SCD operations sometimes span multiple tags, ensuring atomicity is more involved. Note that the access scheduling logic makes the array type transparent to the directory: so long as the SCD array maintains atomicity, forward progress and fairness, it can be used as a drop-in replacement for a sparse array, with no changes to the coherence protocol or controller.

Our SCD implementation satisfies these goals with the following scheduling policies. First, as in conventional sparse directories, concurrent operations to the same address are serialized, and processed in FCFS order, to preserve atomicity and ensure fairness. Second, as in zcaches, the array is pipelined, and we allow concurrent non-conflicting lookups and writes, but only allow one replacement at a time. If the replacement process needs to move or evict a tag from an address of a concurrent request, it waits until that request has finished, to preserve atomicity. Third, similar to prior proposals using Cuckoo hashing where insertions are sometimes on the critical path [59, 96], we introduce an *insertion queue* to avoid the latency introduced by the replacement process. Tags are allocated in the insertion queue first, then inserted in the array. We have observed that, in practice, a 4-entry insertion queue suffices to hide replacement delay for sufficiently provisioned directories, where replacements are short, while severely underprovisioned directories require an 8-entry queue. Finally, to avoid deadlock, operations that require allocating new tags and block on a full insertion queue are not started until they allocate their space. This way, the replacement process is able to move or evict tags belonging to the address of the blocking request.

Performance: With this implementation, operations on a specific address are executed atomically once they start. Operations that require allocating one or more tags (adding a sharer) are considered completed once they have reserved enough space in the insertion queue. Writebacks, which require removing a sharer and never allocate,

are considered complete once the root tag is accessed. Therefore, *adding and removing a sharer are typically as fast in SCD as in a conventional sparse directory.* On the other hand, coherence-induced invalidations on widely shared addresses need to access several leaf tags to retrieve the sharers, invalidate them, and respond to the original requester once all invalidated sharers have responded. This could take longer with SCD than with a sparse directory, where the whole sharer set can be retrieved in one access (e.g., processing 1024 vs 32 sharers/cycle in our example). However, the critical-path latency of invalidations is determined by serialization latency in the network, as the directory can only inject one invalidation request per cycle, so SCD and sparse full-map directories perform similarly. Invalidation delays have a small performance impact in our simulations (Section 5.6), but should they become an issue, they can be reduced by processing invalidations in a hierarchical fashion, using multicast networks, or cruise-missile invalidates [13].

5.3.5 Storage Efficiency

We define *storage efficiency* as the average number of sharers that SCD encodes per tag. Storage efficiency determines how many directory tags are needed, and therefore how to size the directory. When all the lines have a single sharer, SCD has a storage efficiency of 1 sharer/tag. This is a common case (e.g., running a separate workload on each core, or a multithreaded workload where each working set is thread-private and shared code footprint is minimal). When lines have multiple sharers, SCD typically achieves an efficiency higher than 1. For example, using the format in Figure 5.2, a limited pointer tag with three sharers would have a storage efficiency of 3, while a fully shared line would have an efficiency of 1024 sharers/33 tags $\cong$ 31 sharers/tag. If the expected efficiency is consistently higher than 1, one could undersize or power off part of the directory and still achieve negligible invalidations. Note that SCD has a much lower dynamic range of storage efficiency than sparse directories (1-31 sharers/tag vs 1-1024 sharers/tag) but has far fewer sharer bits per tag ($\sim$32 bits vs $\sim$1024 bits).

Although, as we will see in Section 5.6, SCD typically achieves a storage efficiency

≥ 1, its *worst-case efficiency* is smaller than 1. In particular, the worst-case efficiency is 1/2, which happens when a single-sharer line is stored using a two-level bit-vector. Worst-case efficiency decreases as we scale up (e.g., with N-level bit-vectors, it is $1/N$), and might be an issue if the designer wants the directory to provide strict guarantees on evictions (e.g., to avoid interference among applications sharing the CMP). Therefore, we propose two techniques to improve worst-case efficiency.

Line coalescing: A simple optimization is to inspect entries that have few sharers on writebacks, and coalesce them into a limited pointer representation if possible. For example, following Figure 5.2, if every time we remove a sharer and the root bit-vector has two or fewer bits set, we try to coalesce the line, the worst-case efficiency becomes 2/3. If we do this with every line with 3 or fewer sharers, the worst-case efficiency becomes 3/4. Coalescing improves storage efficiency at the expense of additional accesses.

Pointer in root bit-vector: If strict efficiency guarantees are necessary, we can change the tag format to guarantee a worst-case efficiency of 1 by including a single pointer in the root bit-vector tag. When switching a limited pointer tag to a hierarchical bit-vector, the root bit-vector tag keeps one of the sharers in the pointer. If that sharer is removed, one of the sharers represented in the leaf bit-vector tags is moved over to the root tag. With more than two levels, both root and intermediate levels would need to implement the pointer. This guarantees that every tag represents at least one sharer, so the worst-case efficiency is 1. As we will see in Section 5.4, this enables strict guarantees on directory-induced invalidations. However, this format improves storage efficiency at the expense of additional area. For example, using this format in the example in Figure 5.2 would require 45 bits/tag for sharer information instead of 39 to hold the extra pointer (assuming we widen the leaf bit-vectors and narrow the root one).

5.4 Analytical Framework for Directories

We now show that directories built using zcache-like arrays in general, and SCD in particular, can be characterized with analytical models. First, we show that the fraction of replacements that result in evictions and the distribution of lookups per replacement is a function of the directory's occupancy, i.e., the fraction of directory tags used. Second, although directory occupancy is time-varying and workload-dependent, we show that it can be easily bounded. Together, these results show that, with a small amount of overprovisioning, SCD can be designed to guarantee negligible invalidations and high energy efficiency in the worst case.

Uniformity assumption: In our analytical models, we rely on the assumption that the candidates visited in the replacement process have an uniform random distribution over the cache array (Section 3.4.2). We have already shown that, in practice, this is an accurate assumption for both zcache and Vantage. We leverage this assumption in the derivations, and verify its accuracy in Section 5.6 using simulation.

Evictions as a function of occupancy: Assume the directory has T tags, of which U are used. We define directory *occupancy* as $occ = U/T$. Per the uniformity assumption, replacement candidates are independent and uniformly distributed random variables, i.e., $cand_i \sim U[0, T-1]$, and the probability of one being used is $Prob(cand_i\ used) = occ$. If the replacement process is limited to R replacement candidates, the probability that all candidates are being used and we are forced to evict one of them is simply:

$$\begin{aligned} P_{ev}(occ) &= Prob(cand_0\ used \wedge ... \wedge cand_{R-1}\ used) \\ &= Prob(cand_i\ used)^R = occ^R \end{aligned} \tag{5.1}$$

Not surprisingly, this equation has the same form as Equation 3.1, and can also be explained as a particular case of zcache when the replacement policy simply prioritizes unused lines over used lines for eviction. Figure 5.4 plots this probability in linear and semi-logarithmic scales. As in Equation 3.1, with a reasonably large R, the eviction probability quickly becomes negligible. For example, with $R = 64$, $P_{ev}(0.8) = 10^{-6}$,

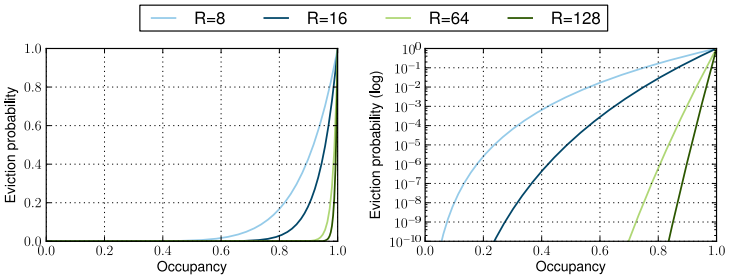


Figure 5.4: Probability that a replacement results in an eviction as a function of occupancy, for R=8, 16, 64 and 128 replacement candidates, in linear and semi-logarithmic-scales.

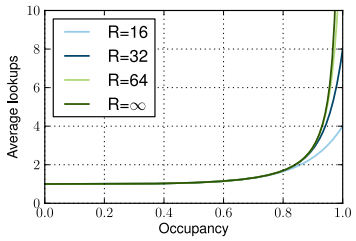


Figure 5.5: Average lookups per replacement as a function of occupancy for a 4-way array.

i.e., only one in a million replacements will cause an eviction when the directory is 80% full.

Lookups per replacement: We now derive the distribution and average number of lookups per replacement as a function of occupancy and the number of ways, W. While Equation 5.1 characterizes worst-case behavior, this illustrates average behavior, and therefore average latency and energy requirements of the directory. First, the probability that all lines are occupied in a single lookup (W ways) is $p = occ^W$. Second, the maximum number of lookups is $L = R/W$. Therefore, the probability of finding an empty line in the k^{th} lookup is $p_k = (1-p)p^{k-1}, k \leq L$. Also, the probability of doing L lookups and not finding any empty line is P_{ev}. Therefore, the average number of lookups is:

$$\begin{aligned} AvgLookups(occ) &= \sum_{k=1}^{L} k(1-p)p^{k-1} + L \cdot P_{ev} \\ &= \frac{1-p^L}{1-p} = \frac{1-occ^R}{1-occ^W} \end{aligned} \tag{5.2}$$

Figure 5.5 plots this value for different numbers of replacement candidates. Fortunately, even for high occupancies, the average number of lookups is much lower than the worst case (R/W). In fact, when evictions are negligible, the average is almost independent of R, and is simply $1/(1-p) = 1/(1-occ^W)$. Therefore, assuming that

we design for a negligible number of evictions, the maximum number of candidates R is irrelevant in the average case. In other words, a reasonable design methodology is to first define the target occupancy based on how much extra storage we want to devote versus how expensive allocation operations are, then set R high enough to satisfy a given eviction probability P_{ev}.

Bounding occupancy: Occupancy is trivially bounded by 1.0 (the directory cannot use more lines than it has). However, if we can bound it to a smaller quantity, we can guarantee a worst-case eviction probability and average lookups independently of the workload. In general, the number of used tags is $U = load/\mathit{eff}$, where $load$ is the number of sharers that need to be tracked, and eff is the storage efficiency. Therefore, $occ = U/T = \frac{load/T}{\mathit{eff}}$. We can bound storage efficiency to $\mathit{eff} \geq 1.0$ sharers/line (Section 5.3.5). With a single-banked directory, the worst-case load is trivially the aggregate capacity of the tracked caches (in lines), which we denote C. Therefore, if we never want to exceed a worst-case occupancy $maxOcc$, we should size the directory with $T = C/maxOcc$ tags. This in turn limits P_{ev} and $AvgLookups$. For example, to ensure that the occupancy never exceeds 90%, we would need to overprovision the directory by 11%, i.e., have 11% more tags than lines are tracked, and with a 4-way, 64-replacement candidate array, this would yield worst-case $P_{ev}(0.9) = 10^{-3}$ and worst-case $AvgLookups(0.9) = 2.9$ lookups/replacement. If we wanted a lower bound on P_{ev} (e.g., to provide stricter non-interference guarantees among competing applications sharing the CMP), we could use $R = 128$, which would give $P_{ev} = 10^{-6}$, and *still require* 2.9 *average lookups*. Furthermore, most applications will not reach this worst-case scenario, and the directory will yield even better behavior. Alternatively, designers can provision the directory for an expected range of occupancies instead of for the worst case, reducing guarantees but saving storage space. In contrast, set-associative directories need to be overprovisioned by 2× or more to reduce evictions, and provide no guarantees [59].

When directories are banked, as it is commonly done with large-scale designs, this bound needs to be relaxed slightly, because the tracked caches will impose a

different load on each bank. If a reasonable hash function is used, distributing addresses uniformly across the K directory banks, from basic probability theory, *load* has a binomial distribution $\sim B(C, 1/K)$, with mean C/K and standard deviation $\sqrt{C/K \cdot (1 - 1/K)}$. Therefore, the lower the number of banks, the more concentrated these values will be around the mean. In the CMP we study ($C = 2^{21}$ lines, $K = 64$ banks), the standard deviation is only 0.5% of its mean, and it can be assumed that the worst-case *load* $\cong C/K$ is constant across banks. In general, both P_{ev} and the number of lookups can be treated as functions of random variable C to determine the exact bounds for a given amount of overprovisioning.

In summary, we have seen that SCD can be characterized with analytical models, and can be tightly sized: high occupancies can be achieved with efficient replacements and incurring a negligible amount of evictions. These models apply to SCD and regular sparse directories implemented with arrays where the uniformity assumption holds (skew-associative caches, zcaches or Cuckoo Directories). We will show that these models are accurate in practice using simulation.

5.5 Experimental Methodology

Modeled system: We use zsim (Section 3.5) to model a large-scale CMP with 1024 cores, shown in Figure 5.6. Table 5.2 summarizes its main characteristics. Each core is in-order and single-threaded, modeled after Atom [62], and has split 32KB L1 instruction and data caches and a private, inclusive, 128KB L2. All cores share a 256MB L3, which is kept coherent using a MESI coherence protocol. The CMP is divided in 64 tiles, each having 16 cores, a directory and L3 bank, and a memory controller. Both L2 and L3 are 4-way zcaches with 16 and 52 replacement candidates, respectively. All directories and all caches except the L1s use H_3 hash functions, which are simple to implement and work well in practice [31]. Tiles are connected with an 8×8 mesh network-on-chip (NoC) with physical express links.

The system we model is in line with several large-scale CMP proposals, such as Rigel [90, 91] and ATAC [102], and represents a reasonable scale-up of commercial designs like Tilera's Gx-3100 [153], which has 100 cores and 32 MB of distributed,

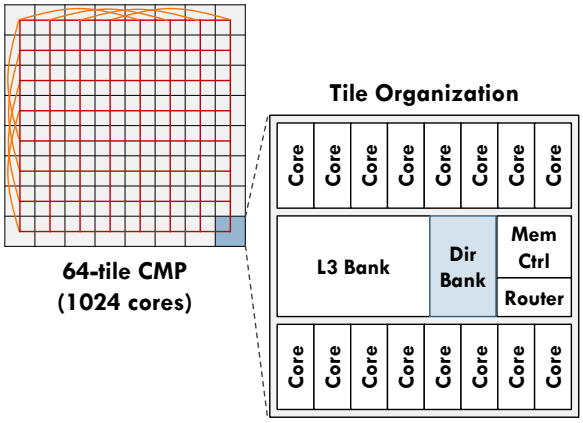


Figure 5.6: Simulated 64-tile, 1024-core CMP: global tile view (including network links) and tile organization.

Cores	1024 cores, x86-64 ISA, in-order, IPC=1 except on memory accesses, 2 GHz
L1 caches	32 KB, 4-way set associative, split D/I, 1-cycle latency
L2 caches	128 KB private per-core, 4-way 16-candidate zcache, inclusive, 5-cycle latency
L3 cache	256 MB NUCA, 64 banks (1 bank/tile), fully shared, 4-way 52-candidate zcache, non-inclusive, 10-cycle bank latency
Global NoC	8×8 mesh with express physical links every 4 routers, 128-bit flits and links, X-Y routing, 2-cycle router traversal, 1-cycle local links, 3-cycle express links
Coherence protocol	MESI protocol, split request-response, no forwards, no silent drops; sequential consistency
Memory controllers	64 MCUs (1 MCU/tile), 200 cycles zero-load latency, 5 GB/s per controller (optical off-chip interface as in [102])

Table 5.2: Main characteristics of the simulated 1024-core CMP.

directory-coherent last-level cache that can be globally shared, and is implemented at 40 nm. We estimate that our target CMP should be implementable at 14 or 11 nm. Using McPAT [108], we find that a scaled-down version of this system with 8 tiles and 128 cores would require 420 mm^2 and 115 W at 32 nm. We use the component latencies of this scaled-down CMP in the 1024-core simulations.

Directory implementations: We compare three different directory organizations: sparse, sparse hierarchical (two-level), and SCD. The classic sparse organization has a full-map 1024-bit sharer vector per line. The hierarchical implementation has a distributed first directory level every two tiles, and a second, banked directory level. Therefore, both levels have 32-bit sharer vectors. SCD has the same organization

as shown in Figure 5.2, with 3 limited pointers and a 32/32 2-level bit-vector organization. All organizations nominally use 4-way zcache arrays with 52 replacement candidates and H_3 hash functions, so the sparse organization is similar to a Cuckoo Directory [59]. All directories are modeled with a 5-cycle access latency. We compare directories with different degrees of *coverage*. Following familiar terminology for TLBs, we define coverage as the maximum number of addresses that can be represented in the directory, as a percentage of the total lines in the tracked caches. Therefore, 100%-coverage Sparse and SCD have as many tags as lines in the tracked caches, while a hierarchical directory with 100% coverage has twice as many tags (as each address requires at least two tags, one per level).

Workloads: We simulate 14 multithreaded workloads selected from multiple suites: PARSEC [16] (blackscholes, canneal, fluidanimate), SPLASH-2 [161] (barnes, fft, lu, ocean, radix, water), SPEC OMP2001 (applu, equake, wupwise), SPEC JBB2005 (specjbb), and BioParallel [84] (svm). We have selected workloads that scale reasonably well to 1024 cores and exhibit varied behaviors in the memory hierarchy (L1, L2 and L3 misses, amount of shared data, distribution of sharers per line, etc.). We simulate complete parallel phases (sequential portions of the workload are fast-forwarded), and report relative execution times as the measure of performance. Runs have at least 200 million cycles and 100 billion instructions, ensuring that all caches are warmed up. We perform enough runs to guarantee stable averages (all results presented have 95% confidence intervals smaller than 1%).

5.6 Evaluation

5.6.1 Comparison of Directory Schemes

Directory size and area: Table 5.3 shows the directory size needed by the different directory organizations (SCD, Sparse, and Hierarchical) for 128 to 1024 cores. We assume line addresses to be 42 bits. Storage is given as a percentage of total tracked cache space. All directories have 100% coverage.

As we can see, SCD significantly reduces directory size. A 2-level SCD uses

Cores	SCD storage	Sparse storage	Hier. storage	Sparse vs SCD	Hier. vs SCD
128	10.94%	34.18%	21.09%	3.12×	1.93×
256	12.50%	59.18%	24.22%	4.73×	1.94×
512	13.87%	109.18%	26.95%	7.87×	1.94×
1024	15.82%	209.18%	30.86%	13.22×	1.95×

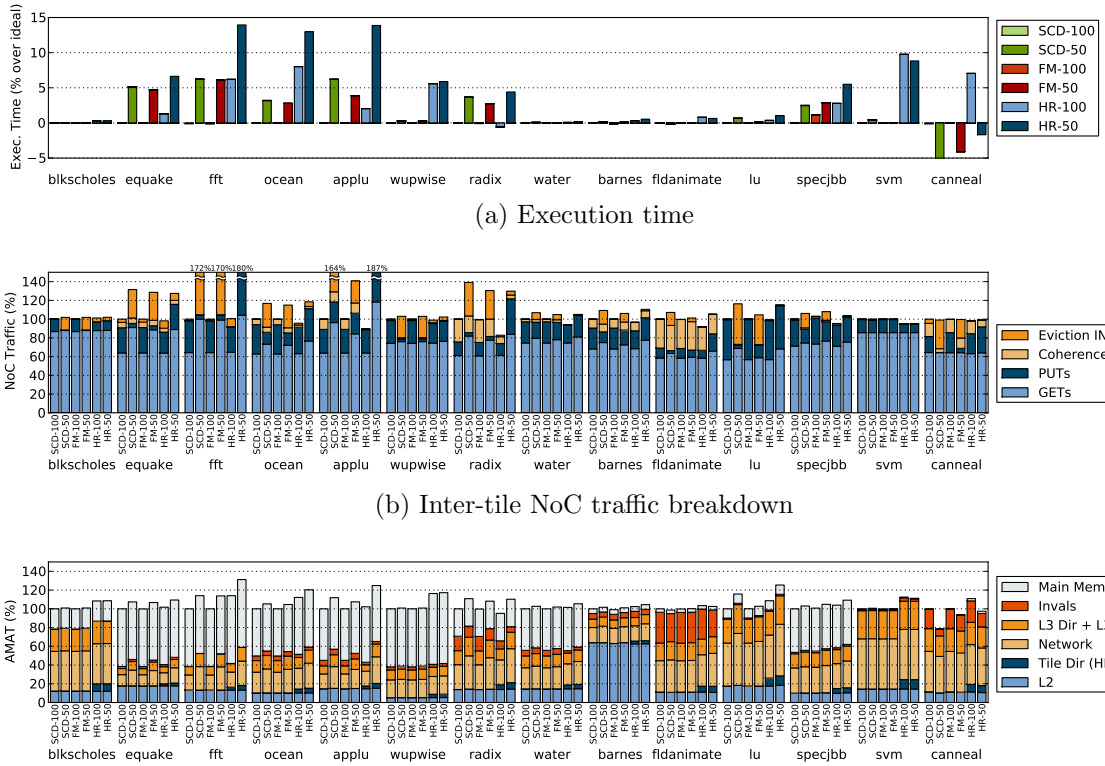
Table 5.3: Directory size requirements for different organizations. Size is given as a percentage of the aggregate capacity of the tracked caches, assuming a 42-bit line address, 64-byte lines and 100% coverage.

3×–13× less space than a conventional sparse directory, and around 2× less than a 2-level hierarchical implementation. A 3-level SCD would be even more efficient (e.g., requiring 18 bits of coherence data per tag instead of 39 at 1024 cores), although gains would be small since the address field would take most of the tag bits.

We can approximate directory area using directory size and assuming the same storage density for the L3 cache and the directory. On our 1024-core CMP, SCD would require 20.2 MB of total storage, taking 3.1% of die area, while a two-level hierarchical directory would require 39.5 MB, taking 6.1% of die area. Sparse directories are basically unimplementable at this scale, requiring 267MB of storage, as much as the L3 cache.

Performance: Figure 5.7 compares execution time, global NoC traffic and average memory access time among different directory organizations. Each directory is simulated at both 100% and 50% coverage. Smaller values are better for all graphs, and results are normalized to those of an idealized directory (i.e., one with no invalidations). Recall that all directory organizations use 4-way/52-candidate zcache arrays. We will discuss set-associative arrays in Section 5.6.4.

Looking at Figure 5.7a, we see that both SCD and Sparse achieve the performance of the ideal directory in all applications when sized for 100% coverage, while their 50%-sized variants degrade performance to varying degrees (except on canneal, which we will discuss later, where performance increases). Underprovisioned Sparse directories perform slightly better than SCD because their occupancy is lower, as they require one line per address. Hierarchical directories, on the other hand, are slightly slower even at 100% coverage, as they require an additional level of lookups, and their



(a) Execution time

(b) Inter-tile NoC traffic breakdown

(c) Average memory access time (AMAT) breakdown

Figure 5.7: Comparison of nominally provisioned (100% coverage) and underprovisioned (50% coverage) directory organizations: SCD, sparse full-map (FM) and 2-level sparse hierarchical (HR). All directories use 4-way/52-candidate zcache arrays.

performance degrades significantly more in the undersized variant. Note that the 50%-coverage Hierarchical directory has about the same area as the 100%-coverage SCD.

Figures 5.7b and 5.7c give more insight into these results. Figure 5.7b breaks down NoC traffic into GET (exclusive and shared requests for data), PUT (clean and dirty writebacks), coherence INV (invalidation and downgrade traffic needed to maintain coherence), and eviction INV (invalidations due to evictions in the directory). Traffic is measured in flits. We see that all the 100%-sized directories introduce practically no invalidations due to evictions, except SCD on canneal, as canneal pushes SCD occupancy close to 1.0 (this could be solved by overprovisioning slightly, as explained in Section 5.4). The undersized variants introduce significant invalidations. This often

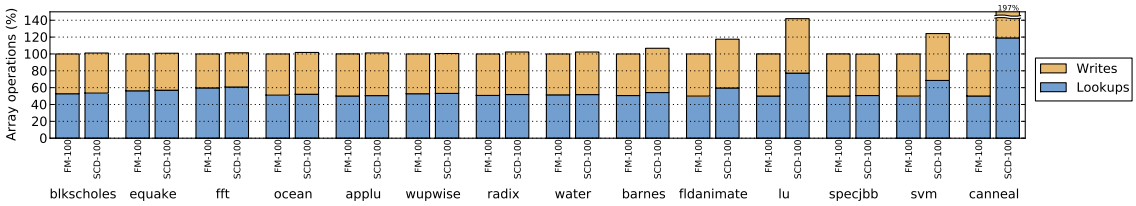


Figure 5.8: Comparison of array operations (lookups and writes) of sparse full-map (FM) and SCD with 100% coverage.

reduces PUT and coherence INV traffic (lines are evicted by the directory before the L2s evict them themselves or other cores request them). However, those evictions cause additional misses, increasing GET traffic. Undersized directories increase traffic by up to 2×. Figure 5.7c shows the effect that additional invalidations have on average memory access time (AMAT). It shows normalized AMAT for the different directories, broken into time spent in the L2, local directory (for the hierarchical organization), NoC, directory and L3, coherence invalidations, and main memory. Note that the breakdown only shows critical-path delays, e.g., the time spent on invalidations is not the time spent on every invalidation, but the critical-path time that the directory spends on coherence invalidations and downgrades. In general, we see that the network and directory/L3 delays increase, and time spent in invalidations decreases sometimes (e.g., in fluidanimate and canneal). This happens because eviction invalidations (which are not on the critical path) reduce coherence invalidations (on the critical path). This is why canneal performs better with underprovisioned directories: they invalidate lines that are not reused by the current core, but will be read by others (i.e., canneal would perform better with smaller private caches). Dynamic self-invalidation [103] could be used to have L2s invalidate copies early and avoid this issue.

In general, we see that hierarchical directories perform much worse when undersized. This happens because both the level-1 directories and level-2 (global) directory cause invalidations. Evictions in the global directory are especially troublesome, since all the local directories with sharers must be invalidated as well. In contrast, an undersized SCD can prioritize leaf or limited pointer lines over root lines for eviction, avoiding expensive root line evictions.

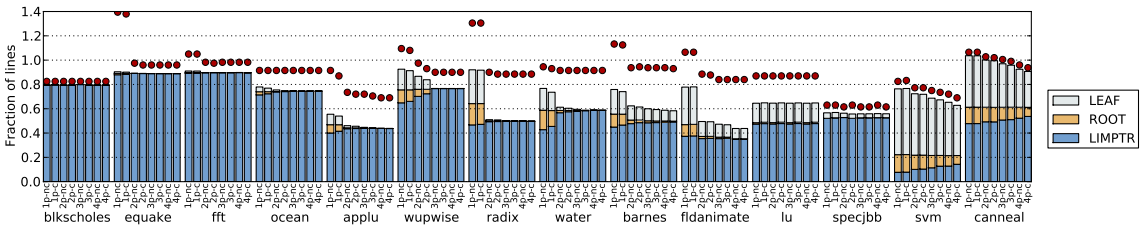


Figure 5.9: Average and maximum used lines as a fraction of tracked cache space (in lines), measured with an ideal SCD directory with no evictions. Configurations show 1 to 4 limited pointers, without and with coalescing. Each bar is broken into line types (limited pointer, root bit-vector and leaf bit-vector). Each dot shows the maximum instantaneous occupancy seen by any bank.

Energy efficiency: Due to a lack of energy models at 11 nm, we use the number of array operations as a proxy for energy efficiency. Figure 5.8 shows the number of operations (lookups and writes) done in SCD and Sparse directories. Each bar is normalized to Sparse. Sparse always performs fewer operations because sharer sets are encoded in a single line. However, SCD performs a number of operations comparable to Sparse in 9 of the 14 applications. In these applications, most of the frequently-accessed lines are represented with limited pointer lines. The only applications with significant differences are barnes (5%), svm, fluidanimate (20%), lu (40%) and canneal (97%). These extra operations are due to two factors: first, operations on multi-line addresses are common, and second, SCD has a higher occupancy than Sparse, resulting in more lookups and moves per replacement. However, SCD lines are narrower, so SCD should be more energy-efficient even in these applications.

5.6.2 SCD Occupancy

Figure 5.9 shows average and maximum used lines in an ideal SCD (with no evictions), for different SCD configurations: 1 to 4 limited pointers, with and without coalescing. Each bar shows average occupancy, and is broken down into the line formats used (limited pointer, root bit-vector and leaf bit-vector). Results are given as a fraction of tracked cache lines, so, for example, an average of 60% would mean that a 100%-coverage SCD would have a 60% average occupancy assuming negligible evictions. These results show the space required by different applications to have negligible

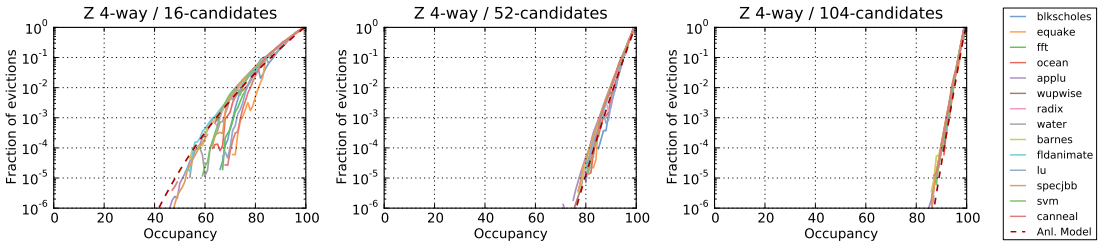


Figure 5.10: Measured fraction of evictions as a function of occupancy, using SCD on 4-way zcache arrays with 16, 52 and 104 candidates, in semi-logarithmic scale. Empirical results match analytical models.

evictions.

In general, we observe that with one pointer per tag, some applications have a significant amount of root tags (which do not encode any sharer), so both average and worst-case occupancy sometimes exceed 1.0×. Worst-case occupancy can go up to 1.4×. However, as we increase the number of pointers, limited pointer tags cover more lines, and root tags decrease quickly (as they are only used for widely shared lines). Average and worst-case occupancy never exceed 1.0× with two or more pointers, showing that SCD's storage efficiency is satisfactory. Coalescing improves average and worst-case occupancy by up to 6%, improving workloads where the set of shared lines changes over time (e.g., water, svm, canneal), but not benchmarks where the set of shared lines is fairly constant (e.g., fluidanimate, lu).

5.6.3 Validation of Analytical Models

Figure 5.10 shows the measured fraction of evictions (empirical P_{ev}) as a function of occupancy, on a semi-logarithmic scale, for different workloads. Since most applications exercise a relatively narrow band of occupancies for a specific directory size, to capture a wide range of occupancies, we sweep coverage from 50% to 200%, and plot the average for a specific occupancy over multiple coverages. The dotted line shows the value predicted by the analytical model (Equation 5.1). We use 4-way arrays with 16, 52 and 104 candidates. As we can see, the theoretical predictions are accurate in practice.

Figure 5.11 also shows the average number of lookups for the 52-candidate array,

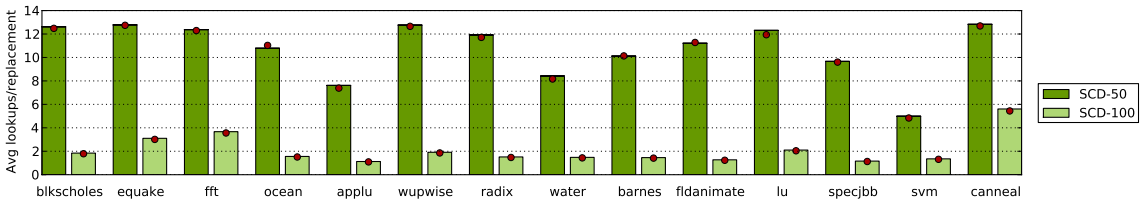


Figure 5.11: Average lookups per replacement on a 4-way, 52-candidate array at 50% and 100% coverage. Each bar shows measured lookups, and the red dot shows the value predicted by the analytical model. Empirical results match analytical models, and replacements are energy-efficient with sufficiently provisioned directories.

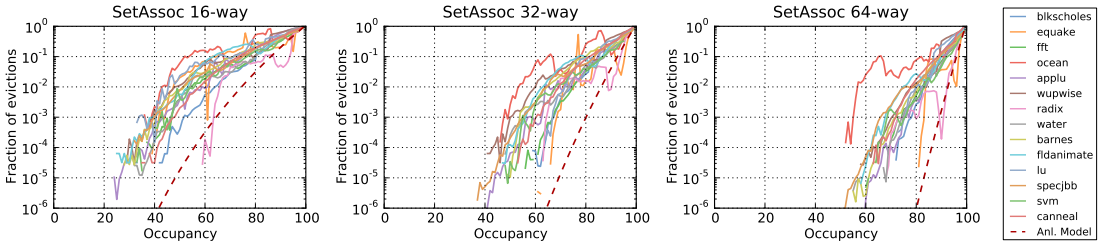


Figure 5.12: Measured fraction of evictions as a function of occupancy, using SCD on set-associative arrays with 16, 32 and 64 ways, in semi-logarithmic scale.

sized at both 50% and 100% coverage. Each bar shows the measured lookups, and the red dot shows the value predicted by the analytical model. Again, empirical results match the analytical model. We observe that with a 100% coverage, the number of average lookups is significantly smaller than the maximum ($R/W = 13$ in this case), as occupancy is often in the 70%-95% range. In contrast, the underprovisioned directory is often full or close to full, and the average number of lookups is close to the maximum.

In conclusion, we see that SCD's analytical models are accurate in practice. This lets architects size the directory using simple formulas, and enables providing strict guarantees on directory-induced invalidations and energy efficiency with a small amount of overprovisioning, as explained in Section 5.4.

5.6.4 Set-Associative Caches

We also investigate using SCD on set-associative arrays. Figure 5.12 shows the fraction of evictions as a function of occupancy using 16, 32 and 64-way caches. All designs use H_3 hash functions. As we can see, set-associative arrays do not achieve the analytical guarantees that zcaches provide: results are both significantly worse than the model predictions and application-dependent. Set-associative SCDs incur a significant number of invalidations even with a significantly oversized directory. For example, achieving $P_{ev} = 10^{-3}$ on these workloads using a 64-way set-associative design would require overprovisioning the directory by about 2×, while a 4-way/52-candidate zcache SCD needs around 10% overprovisioning. In essence, this happens because set-associative arrays violate the uniformity assumption, leading to worse associativity than zcache arrays with the same candidates.

These findings essentially match those of Ferdman et al. [59] for sparse directories. Though not shown, we have verified that this is not specific to SCD — the same patterns can be observed with sparse and hierarchical directories as well. In conclusion, if designers want to ensure negligible directory-induced invalidations and guarantee performance isolation regardless of the workload, directories should not be built with set-associative arrays. Note that using zcache arrays has more benefits in directories than in caches. In caches, zcaches have the latency and energy efficiency of a low-way cache on hits, but replacements incur similar energy costs as a set-associative cache of similar associativity (Section 3.3.2). In directories, the cost of a replacement is also much smaller since replacements are stopped early.

5.6.5 Replacement Policy

Figure 5.13 shows the execution time when using an underprovisioned 52-candidate SCD under different replacement policies. Execution times are normalized to an ideal SCD with no evictions. We only show underprovisioned directories because well-provisioned directories show negligible evictions, and the replacement policy is irrelevant. We compare LRU, NumSharers, Cuckoo and Random policies. LRU is implemented with 8-bit coarse-grain timestamps as described in Section 3.3.5.

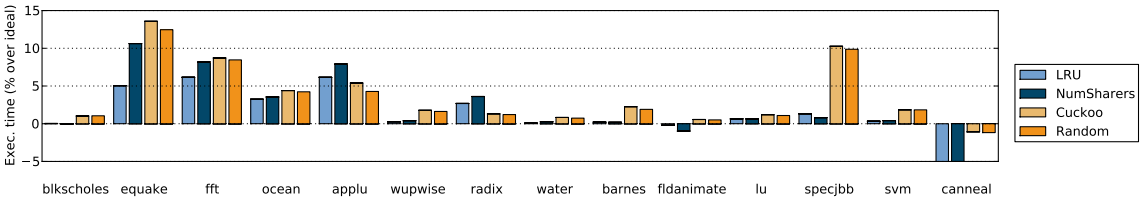


Figure 5.13: Execution time penalty for using an underprovisioned SCD (50% coverage) under several replacement policies. Results are normalized to an ideal SCD (no evictions).

NumSharers simply prioritizes candidates by their number of sharers, conservatively overestimating the number of sharers of a root line as the number of bits set times the leaf bit-vector size. NumSharers requires no extra storage, but needs a small amount of logic. Cuckoo models the replacement process of Cuckoo Directories instead of zcaches, where candidates are obtained depth-first and the line displaced at the maximum move threshold is evicted, as explained in Section 5.2. Finally, Random simply selects a candidate at random.

Although performance varies by application, we can make several observations. First, LRU generally outperforms the other policies, so if designers are to implement an underprovisioned directory (e.g., for an application-specific accelerator where typical sharing patterns are known in advance to not stress the directory beyond a certain capacity), it may be desirable to invest in additional state for the replacement policy. Second, NumSharers performs well when there is a high spread in the number of sharers (e.g., svm, lu), but performs like Random replacement when most data is private (e.g., equake, fft). Finally, Cuckoo replacement performs equivalently to Random, which makes it sub-optimal with underprovisioned directories.

5.7 Summary

We have presented SCD, a single-level, scalable coherence directory design that is area-efficient, energy-efficient, requires no modifications to existing coherence protocols, represents sharer sets exactly, and incurs a negligible number of invalidations. SCD exploits the insight that directories need to track a fixed number of sharers,

not addresses, by representing sharer sets with a variable number of tags: lines with one or few sharers use a single tag, while widely shared lines use additional tags. SCD uses efficient highly-associative caches that allow it to be characterized with simple analytical models, and enables tight sizing and strict probabilistic bounds on evictions and energy consumption. SCD requires 13× less storage than conventional sparse full-map directories at 1024 cores, and is 2× smaller than hierarchical directories while using a simpler coherence protocol. Using simulations of a 1024-core CMP, we have shown that SCD achieves the predicted benefits, and its analytical models on evictions and energy efficiency are accurate in practice.

Chapter 6

GRAMPS: Dynamic Fine-Grain Scheduling of Irregular Data, Task and Pipeline Parallelism

6.1 Introduction

Large-scale CMPs require abundant parallelism, but most programmers find it hard to expose enough parallelism using conventional low-level techniques. This has created a renewed interest in high-level parallel programming models such as Cilk [60], TBB [81], CUDA [122], OpenCL [94], and StreamIt [151]. These models provide constructs to express parallelism and synchronization in a safe and manageable way, and their runtimes take care of resource management and scheduling for the programmer. However, for this approach to succeed, we need efficient parallel runtimes and schedulers. Unfortunately, as Section 2.3.1 discusses, previously proposed scheduling techniques have significant drawbacks except with a restricted set of programming models. On one hand, dynamic schedulers (such as Task-Stealing or Breadth-First) work well on programs with no or simple dependencies (e.g., fork-join), but they produce inefficient schedules and cannot bound memory footprint under more complex dependencies (e.g., pipeline-parallel programs). On the other hand, Static schedulers handle programs with complex dependencies well, but do not admit run-time load

balancing, so variability introduced by the application or the underlying hardware causes load imbalance, hindering performance.

In this chapter, we present a scheduler implementation for pipeline-parallel programs that performs fine-grain dynamic load balancing efficiently. Specifically, we implement the first real runtime for GRAMPS [146], a programming model that focuses on supporting applications with irregular task, pipeline and data parallelism (in contrast to classical stream programming models and schedulers, which require programs to be regular). GRAMPS applications are expressed as a graph of stages that communicate either explicitly through data queues or implicitly through memory buffers. Compared to programming models with simpler semantics (such as Task-Stealing), knowing the application graph gives two main benefits. First, the graph contains all the producer-consumer relationships, enabling improved locality. Second, memory footprint is easily bounded by limiting the size of queues and memory buffers. However, prior GRAMPS work [146] was based on an idealized simulator with no scheduling overheads, making it an open question whether a practical GRAMPS runtime could be designed. To this end, this chapter presents the following contributions:

1. We present the first real implementation of a GRAMPS runtime for multi-core machines. The scheduler introduces two novel techniques. First, *task-stealing with per-stage queues* and a *queue backpressure* mechanism enable dynamic load balancing while maintaining bounded footprint. Second, a *buffer management* technique based on *packet-stealing* enables dynamic allocation of data packets at low overhead, while maintaining good locality even in the face of frequent producer-consumer communication. To our knowledge, this is the first runtime that supports dynamic fine-grain scheduling of irregular streaming applications. While our runtime is specific to GRAMPS, the techniques used can be applied to other streaming programming models, such as StreamIt [151] or Delite [26]. We evaluate this runtime on a variety of benchmarks using a modern multi-core machine, and find that it efficiently schedules both simple and complex application graphs while preserving locality and bounded footprint.
2. We compare GRAMPS with commonly used scheduling techniques. Since the

GRAMPS programming model provides a superset of the constructs of other models, the runtime can work with the other families of schedulers. We implement these schedulers and compare them using the same infrastructure, allowing us to focus on the differences between schedulers, not runtime implementations. We find that Task-Stealing is generally a good approach to schedule simple graphs, but becomes inefficient with complex graphs or ordered queues, and does not guarantee bounded footprint in general. Breadth-First scheduling is simple, but does not take advantage of pipeline parallelism and requires significantly more footprint than other approaches, putting more pressure on the memory subsystem. Finally, Static scheduling provides somewhat better locality than schedulers using dynamic load balancing due to carefully optimized, profile-based schedules. However, this benefit is negated by significant load imbalance, both from application irregularities and the dynamic nature of the underlying hardware. We show that our proposed GRAMPS scheduler achieves significant benefits over each of these approaches.

6.2 Background on Scheduling Techniques

In this section, we give the necessary background and definitions for different scheduling approaches: Task-Stealing, Breadth-First, Static, and GRAMPS. Rather than comparing specific scheduler implementations, our objective is to distill the key scheduling policies of each and to compare them.

6.2.1 Scheduler Features

We use four main criteria to compare scheduling approaches:

- **Support for shaders**: The scheduler supports a built-in construct for *data-parallel* work, which is automatically parallelized by the scheduler across independent lightweight instances.
- **Support for producer-consumer**: The scheduler is aware of data produced as intermediate results (i.e., created and consumed during execution) and attempts

Approach	Supports Shader	Producer-Consumer	Hierarchical Work	Adaptive Schedule	Examples
Task-Stealing	No	No	No	Yes	Cilk, TBB, OpenMP
Breadth-First	Yes	No	Yes	No	CUDA, OpenCL
Static	Yes	Yes	Yes	No	StreamIt, Imagine
GRAMPS	**Yes**	**Yes**	**Yes**	**Yes**	GRAMPS

Table 6.1: Comparison of different scheduling approaches.

to exploit this during scheduling.

- **Hierarchical work**: The scheduler supports work being expressed and grouped at different granularities rather than all being expressed at the finest granularity.
- **Adaptive schedule**: The scheduler has freedom to choose what work to execute at run-time, and can choose from all available work to execute.

Using the above criteria, we discuss the four scheduling approaches considered. Table 6.1 summarizes the differences between scheduling approaches.

6.2.2 Previous Scheduling Approaches

Task-Stealing: A Task-Stealing scheduler sees an application as a set of explicitly divided, concurrent and independent tasks. These tasks are scheduled on worker threads, where each worker thread has a queue of ready tasks to which it enqueues and dequeues tasks. When a worker runs out of tasks, it tries to steal tasks from other workers.

Task-Stealing has been shown to impose low overheads and scale better than alternative task pool organizations [76]. Therefore, it is used by a variety of parallel programming models, such as Cilk [60], X10 [34], TBB [81], OpenMP [53], and Galois [99].

Task-Stealing is rooted in programming models that exploit fine-grain parallelism, and often focus on low-overhead task creation and execution [10]. As a result, they tend to lack features that add overhead, such as task priorities. All tasks appear to be equivalent to the scheduler, preventing it from exploiting producer-consumer relationships. Task-Stealing has several algorithmic options that provide some control over scheduling, such as the order in which tasks are enqueued, dequeued and stolen

(e.g., FIFO or LIFO) or the choice of queues to steal from (e.g., randomized or nearest neighbor victims). Several programming models, such as Cilk and X10, focus on fork-join task parallelism. In these cases, LIFO enqueues and dequeues with FIFO steals from random victims is the most used policy, as it achieves near-optimal performance and guarantees that footprint grows at most linearly with the number of threads [19]. However, several studies have shown that there is no single best scheduling policy in the general case [53, 67, 76]. In fact, Galois, which targets irregular data-parallel applications that are often sensitive to the scheduling policy, exposes a varied set of policies for task grouping and ordering, and enables the programmer to control the scheduling policy [99, 119].

In this work we leverage Task-Stealing as an efficient load balancing mechanism, combining it with additional techniques to schedule pipeline-parallel applications efficiently.

Breadth-First: In Breadth-First scheduling, the application is specified as a sequence of *data-parallel stages* or *kernels*. A stage is written in an implicitly parallel style that defines the work to be performed per input element. Stages are executed one at a time, and the scheduler automatically instances and manages a collection of shaders that execute the stage, with an implicit barrier between stages.

This model is conceptually very simple, but has weaknesses in extracting parallelism and constraining data footprint. If an application has limited parallelism per stage but many independent stages, the system will be under-utilized. Furthermore, even if a stage produces results at the same rate as the next stage that will consume them, the explicit barrier leaves no alternative but to spend memory space and bandwidth to spill the entire intermediate output of the first stage and to read it back during the second stage.

GPGPU programming models (e.g., CUDA [122] and OpenCL [94]) rely on a GPU's high bandwidth and large execution context count to implement Breadth-First schedulers. However, such assumptions could be problematic for a general-purpose multi-core machine.

Static: In this scheduling approach, an application is expressed as a graph of *stages*, which communicate explicitly via data *streams*. The scheduler uses static analysis, profiling, and/or user annotations to derive the execution time of each stage and the communication requirements across stages. Using this knowledge, it schedules stages across execution contexts in a pattern optimized for low inter-core communication and small memory footprints [64, 88, 98, 125]. Scheduling is done offline, typically by the compiler, eliminating run-time scheduling overheads.

Static schedulers take advantage of producer-consumer locality by scheduling producers and consumers in the same or adjacent cores. They work well when all stages are regular, but cannot adapt to irregular or data-dependent applications.

This scheduling approach is representative of StreamIt [151] and streaming architectures [48, 92], where it is assumed that a program has full control of the machine. However, it can suffer load imbalance in general-purpose multi-cores where resources (e.g., cores or memory bandwidth available to the application) can vary at run-time, as we will see in Section 6.7.

6.3 The GRAMPS Programming Model

We now discuss the core concepts of the GRAMPS programming model that are relevant to scheduling. However, GRAMPS is expressive enough to describe a wide variety of computations. A full description of all constructs supported by the GRAMPS programming model and its detailed API can be found in [145].

GRAMPS applications are structured as graphs of application-defined *stages* with producer-consumer communication between stages through *data queues*. Application graphs may be pipelines, but cycles are also allowed. Figure 6.1 shows an example application graph.

The GRAMPS programming model defines two types of stages: *Shaders* and *Threads* (there are also Fixed-function stages, but they are effectively Thread stages implemented in hardware and not relevant to this paper). Shader stages are stateless and automatically instanced by the scheduler. They are an efficient mechanism to express data-parallel regions of an application. Thread stages are stateful, and must

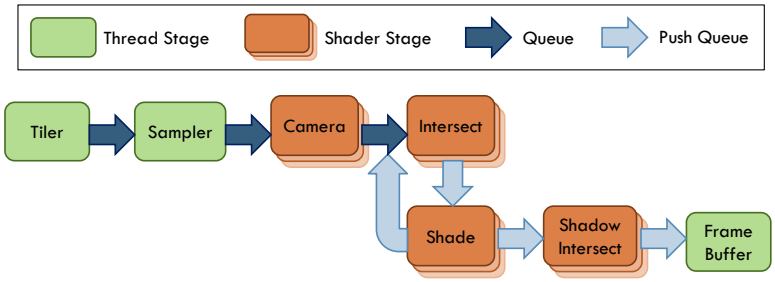


Figure 6.1: A GRAMPS application: the raytracing graph from [146].

be manually instanced by the application. Thread stages are typically used to implement task-parallel, serial, and other regions of an application characterized by (1) large per-element working sets or (2) operations dependent on multiple elements at once (e.g., reductions or re-sorting of data).

Stages operate upon data queues in units of *packets*, which expose bundles of grouped work over which queue operations and runtime decisions can be amortized. The application specifies the capacity of each queue in terms of packets and whether GRAMPS must maintain its contents in strict FIFO order. Applications can also use *buffers* to communicate between stages. Buffers are statically sized random-access memory regions that are well suited to store input datasets and final results.

There are three basic operations on queues: `reserve`, `commit`, and `push`. `reserve` and `commit` claim space in a queue and notify the runtime when the stage is done with it (either input was consumed or output was produced). Thread stages explicitly use `reserve` and `commit`. For Shader stages, GRAMPS implicitly `reserve`s packets before running a shader and `commit`s them when it finishes. `push` provides support for shaders with variable output. Shaders can `push` elements to a queue instead of whole packets. These elements are buffered and coalesced into full packets by the runtime, which then enqueues them. For example, in Figure 6.1 the Shadow Intersect stage operates on 32-ray input packets using SIMD operations, but the Shade stage produces a variable number of output rays. Push queues allow full 32-ray packets to be formed, maintaining the efficiency of SIMD operations.

Queue sets provide a mechanism to enable parallel consumption with synchronization: packets are consumed in sequence within each *subqueue*, but different subqueues

may be processed in parallel. Consider a renderer updating its final output image: with unconstrained parallelism, instances cannot safely modify pixel values without synchronization. If the image is divided into disjoint tiles and updates are grouped by tile, then tiles can be updated in parallel. In Figure 6.1, by replacing the input queue to the frame buffer stage with a queue set, the stage can be replaced with an instanced Thread stage (with one instance per subqueue) to exploit parallelism.

A GRAMPS scheduler should dynamically multiplex Thread and Shader stage instances onto available hardware contexts. The application graph can be leveraged to (1) reduce footprint by giving higher priority to downstream stages, so that the execution is geared towards pulling the data out of the pipeline, (2) bound footprint strictly by enforcing queue sizes, and (3) exploit producer-consumer locality by co-scheduling producers and consumers as much as possible.

At a high level, GRAMPS and streaming programming models have similar goals: both attempt to minimize footprint, exploit producer-consumer locality, and load-balance effectively across stages. However, GRAMPS achieves these goals via dynamic scheduling at run-time, while Static scheduling performs it offline at compile-time. GRAMPS also dynamically emulates filter fusion and fission [64] by co-locating producers and consumers on the same execution context and by time-multiplexing stages. Most importantly, Static schedulers rely on regular stage execution times and input/output rates to derive the long-running *steady state* of an application, which they can then schedule [104]. In contrast, GRAMPS does not require applications to have a steady state, allowing dynamic or irregular communication and execution patterns. For instance, a thread stage can issue an impossibly large `reserve`, which will be satisfied only when all the upstream stages have finished, thus effectively forming a barrier.

6.4 GRAMPS Runtime Implementation

Task-based schedulers, such as Task-Stealing, can perform load balancing efficiently because they represent work in compact tasks, so the cost of moving a task between cores is significantly smaller than the time it takes to execute it. However, these

schedulers do not include support for data queues. On the other hand, in Static streaming schedulers worker threads simply run through a pre-built schedule, and have no explicit notion of a task. Work is implicitly encoded in the availability of data in each worker's fixed-size input buffers. Since *scheduling and buffer management are so fundamentally bound*, fine-grain load balancing on streaming runtimes is unachievable.

To achieve fine-grain load balancing and bounded footprint, the GRAMPS runtime *decouples scheduling and buffer management*. The runtime is organized around two entities:

- A *scheduler* that tracks runnable tasks and decides what to run on each thread context.
- A *buffer manager* that allocates and releases packets. It is essentially a specialized memory allocator for packets.

6.4.1 Scheduler Design

The GRAMPS scheduler is task-based: at initialization, the scheduler creates a number of *worker threads* using PThread facilities. Each of these threads has *task queues* with priorities, to which it enqueues newly produced tasks and from which it dequeues tasks to be executed. As in regular task-stealing, when a worker runs out of tasks, it tries to obtain more by stealing tasks from other threads. Worker threads leverage the application graph to determine the order in which to execute and steal tasks. All these operations are performed in a scalable but globally coordinated fashion.

Figure 6.2 shows an overview of the scheduler organization. We begin by describing how different kinds of stages are represented and executed in the runtime. We then describe the scheduling algorithms in detail.

Shader Stages: Shader stages are stateless and data-parallel, and a Shader can always be run given a packet from its input queue. Therefore, every time an input packet for a Shader stage is produced, a new Shader task with a pointer to that packet is generated and enqueued in the task queue. Shaders cannot block, so they are executed non-preemptively, avoiding context storage and switching overheads.

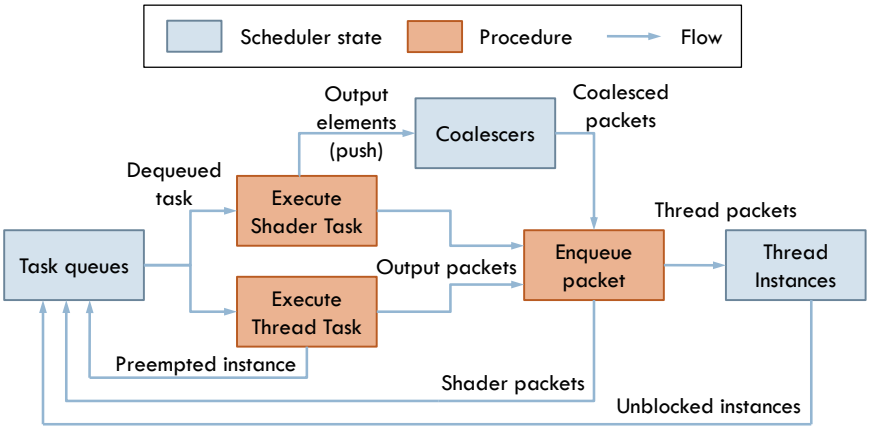


Figure 6.2: Scheduler organization and task/packet flows.

Shaders have two types of output queues: *packet queues*, to which they produce full packets, and *push queues*, to which they enqueue element by element. Executing a Shader with no push queues is straightforward: first, the output packets (or packet) are allocated by the buffer manager. The Shader task is then executed, its input packet is released to the buffer manager, and the generated packets are made available, possibly generating additional tasks. Shaders with push queues are treated slightly differently: each worker thread has a *coalescer*, which aggregates the elements enqueued by several instances of the Shader into full packets, and makes the resulting packets available to the scheduler.

The GRAMPS runtime and API are designed to avoid data copying: application code has direct access to input packets, and writes to output packets directly, even with push queues.

Thread Stages: Thread stages are stateful, long-lived, and may have queue ordering requirements. In particular, Thread stages operate on input and output queues explicitly, `reserving` and `committing` packets from them. Thus, they need to be executed preemptively: they can either block when they try to `reserve` more packets than are available in one of their input queues, or the scheduler can decide to preempt them at any `reserve` or `commit` operation.

To facilitate low-cost preemption, each Thread stage instance is essentially implemented as an *user-level thread*. We encapsulate all its state (stack and context)

and all its input queues into an *instance* object. Input queues store packets in FIFO order.

Each task for a Thread stage represents a *runnable* instance, and is simply a pointer to the instance object. At initialization, an instance is always runnable. A Thread stage instance that is preempted by the runtime before it blocks is still runnable, so after preemption we re-enqueue the task. To amortize preemption overheads, the runtime allows an instance to produce a fixed number of output packets (currently 32) before it preempts it. However, when an instance blocks, it is not runnable, so no task is generated. Instead, when enough packets are enqueued to the input queue that the instance blocked on, the runtime unblocks the instance and produces a task for it.

Note that by organizing scheduler state around tasks, coalescers, and instance objects, *only data queues that feed Thread stages actually exist as queues.* Queues that feed Shader stages are a useful abstraction for the programming model, but they do not exist in the implementation: each packet in these logical queues is physically stored as a task in the task queues. This is important, because many applications perform most of the work in Shaders, so this organization bypasses practically all the overheads of having data queues.

Per-Stage Task Queues: As mentioned in Section 6.2, GRAMPS gives higher priority to stages further down the pipeline. Executing higher priority stages drains the pipeline, reducing memory footprint.

To implement per-stage priorities, each worker thread maintains a set of task queues, one per stage. For regular dequeues, a task is dequeued from higher priority stages first. When stealing tasks, however, the GRAMPS scheduler steals from lower priority stages first, which produce more work per invocation: these stages are at a lower depth in the application graph, so they will generate additional tasks enqueued to their consumer stages, which is desirable when threads are running out of tasks. For example, in Figure 6.1, a Camera task will generate more work than a Shadow Intersect task. To keep task queue operation costs low on graphs with many stages, worker threads store pointers to the highest and lowest queues that have tasks. The

range of queues with tasks is small (typically 1 to 3), so task dequeues are fast.

We implement each task queue as a Chase-Lev deque [35], with LIFO local enqueues/dequeues and FIFO steals. Local operations do not require atomic instructions, and steals require a single atomic operation. Stealing uses the non-blocking ABP protocol [10] and is locality-aware, as threads try to steal tasks from neighboring cores first.

Footprint and Backpressure: In task-based parallel programming models, it is generally desirable to guarantee that footprint is bounded regardless of the number of worker threads. Programming models that exploit fork-join parallelism, like Cilk and X10, can guarantee that footprint grows at most linearly with the number of threads by controlling the queuing and stealing policies [4, 19]. However, most pipeline-parallel and streaming programming models cannot limit footprint as easily. In particular, GRAMPS applications can experience unbounded memory footprint in three cases:

1. Bottlenecks on Thread consumers: If producers produce packets faster than a downstream Thread stage can consume them, an unbounded number of packets can be generated.
2. Thread preemption policy: In order to amortize preemption overheads, GRAMPS does not immediately preempt Thread stages. However, without immediate preemption, footprints can grow superlinearly with the number of worker threads due to stealing [19].
3. Cycles: A graph cycle that generates more output than input will consume unbounded memory regardless of the scheduling strategy.

To solve issues (1) and (2), the runtime enforces bounded queue sizes by applying *backpressure*. The runtime tracks the utilization of each data queue in a scalable fashion, and when a queue becomes full, the stages that output to that queue are marked as non-executable. This guarantees that both data queue and task queue footprints are bounded.

In general, guaranteeing bounded footprint and deadlock-freedom with cycles is not a trivial task, and Static scheduling algorithms have significant problems handling

cycles [98, 125]. To address this issue, we do not enforce backpressure on *backward* queues (i.e., queues that feed a lower-priority stage from a higher-priority stage). If the programmer introduces cycles with uncontrolled loops, we argue that this is an incorrect GRAMPS application, similar to reasoning that a programmer will not introduce infinite recursion in a Cilk program. If a cycle is well behaved, it is trivial to see that footprint is bounded due to stage priorities, even with stealing.

Overall, we find that backpressure strictly limits the worst-case footprint of applications while adding minimal overheads. We will evaluate the effectiveness of this approach in Section 6.7.

Ordered Data Queues: Except for push queues, GRAMPS queues can be FIFO-ordered, which guarantees that the consuming stage of a queue will receive packets in the same order as if the producing stage was run serially. FIFO ordering on queues between Thread stages is maintained by default, but queues with Shader inputs or outputs are not automatically ordered since Shader tasks can execute out of order. Guaranteeing queue ordering allows GRAMPS to implement ordering-dependent streaming applications, but a naïve implementation could cause significant overheads.

We leverage the fact that guaranteeing ordering across two Thread stages is sufficient to guarantee overall ordering. Specifically, to maintain ordering on a chain of stages with Thread stage endpoints and Shaders in the middle, the runtime allocates and enqueues the output packet of the leading Thread stage and the corresponding input packet of the last stage atomically. The pointer to that last packet is then propagated through the intermediate Shader packets. While Shaders can execute out of order, at the end of the chain the packet is filled in and made available to the Thread consumer in order. Essentially, the last stage's input queue acts as a reorder buffer.

This approach minimizes queue manipulation overheads, but it may increase footprint since packets are pre-reserved and, more importantly, the last queue is subject to head-of-line blocking. To address this issue, task queues of intermediate Shader stages follow FIFO ordering instead of the usual LIFO.

6.4.2 Buffer Manager Design

To decouple scheduling and buffer management, we must engineer an efficient way for the scheduler to dynamically allocate new packets and release used ones, while preserving locality. This is the function of the *buffer manager*.

The simplest possible buffer manager, if the system supports dynamic memory allocation, is to allocate packets using malloc and release them using free. We call this a *dynamic memory buffer manager*. However, this approach has high overheads: as more threads stress the memory allocator, synchronization and bookkeeping overheads can quickly dominate execution time.

A better buffer management strategy without dynamic memory overheads is to use per-queue, statically sized memory pools, and allocate packets from the corresponding queue. Since queue space is bounded in GRAMPS, we can preallocate enough buffer space per queue, and avoid malloc/free calls and overhead completely. We refer to this approach as *per-queue buffer manager*. However, this approach may hurt locality, as different threads share the same buffer space. To maximize locality, we would like a packet to be reused by the same core as much as possible. Nevertheless, it would be inefficient to just partition each per-queue pool among workers, since a worker's buffer space demands are not known in advance, and often change throughout execution. Additionally, accessing the shared queue pools can incur synchronization overheads.

Instead, we propose a specialized *packet-stealing buffer manager* that *maximizes locality* while maintaining *low overheads*. In this scheme, queues get their packets from a set of pools, where each pool contains all the packets of the same size. Initially, within each pool, packets are evenly divided across worker threads; each allocation tries to dequeue a packet from the corresponding partition. However, if a worker thread finds its partition empty, it resorts to stealing to acquire additional packets. When done using a packet, a worker thread enqueues the packet back to its partition. To amortize stealing overheads, threads steal multiple packets at once.

Since backpressure limits queue size, stealing is guaranteed to succeed, except when cycles are involved. For cycles, the worker tries one round of stealing, and if it fails, it allocates new packets to the pool using malloc. In practice, this does not happen if loop queues are sized correctly, and is just a deadlock avoidance safeguard.

Packet-stealing keeps overheads low, and more importantly, enables high reuse. For example, in applications with linear pipelines, the LIFO policy will cause each worker thread to use only two packets to traverse the pipeline. As a result, packet stealing only happens as frequently as stealing in the task queues takes place, which is rare for balanced applications.

6.5 Other Scheduling Approaches

As mentioned in Section 6.1, we augment our GRAMPS implementation to serve as a testbed for comparing other scheduling approaches. Specifically, we have defined a modular scheduler interface that enables using different schedulers.

We preserve the core GRAMPS abstractions and APIs —data queues, application graphs, etc.— for all schedulers, even for those used in programming models that customarily lack built-in support for them. This isolates the changes derived from scheduling policies from any distortion caused by changing the programming model. Additionally, it eliminates application implementation variation, as the same version of each application is used in all four modes.

The rest of this section describes the specific designs we chose to represent their respective scheduling approaches. While several variations are possible for a given scheduler type, we strive to capture the key philosophy behind each scheduler, while leaving out implementation-specific design choices.

6.5.1 Task-Stealing Scheduler

Our Task-Stealing scheduler mimics the widely used Cilk 5 scheduler [60]. It differs from the GRAMPS scheduler in three key aspects. First, each worker thread has a single LIFO Chase-Lev task queue [35]; i.e., there are no per-stage queues. Steals are done from the tail of the queue. Second, data queues are unbounded (without per-stage task queues, we cannot enforce backpressure). Third, each thread stage is preempted as soon as it commits a single output packet. This emulates Cilk's *work-first* policy, which switches to a child task as soon as it is created. This policy enables

Cilk to guarantee footprint bounds [60]. Although this does not imply bounded buffer space for GRAMPS, the work-first approach works well in limiting footprint except when there is contention on thread consumers, as we will see in Section 6.7.

6.5.2 Breadth-First Scheduler

The Breadth-First scheduler executes one stage at a time in breadth-first order, so a stage is run only when all its producers have finished. It is the simplest of our schedulers, and represents a typical scheduler for GPGPU programming model (e.g., CUDA). All worker threads run the current stage until they are out of work, then advance in lock-step to the next stage. As usual, load balancing is implemented on each stage with Chase-Lev dequeues. Some of our applications have cycles in their graphs so the scheduler will reset to the top of the graph a finite number of times if necessary. As with Task-Stealing, Breadth-First has no backpressure.

6.5.3 Static Scheduler

The Static scheduler represents schedulers for stream programming models [50, 151]. To generate a static schedule, the application is first profiled running under the GRAMPS scheduler with the desired number of worker threads. We then run METIS [89] to compute a graph partitioning that minimizes the communication to computation ratio, while balancing the computational load in each partition. We then feed the partitioning to the Static scheduler, which executes the application again following a minimum-latency schedule [88], assigning each partition to a hardware context.

The Static scheduler does no load balancing; instead, worker threads have producer-consumer queues that they use to send and receive work [64, 98, 125]. Once a thread runs out of work, it simply waits until it receives more work or the phase terminates. To handle barriers in programs with multiple phases, the Static scheduler produces one schedule per phase.

Overall, the Static scheduler trades off dynamic load balancing for better locality and lower communication.

Workload	Origin	Graph Complexity	Shader Work	Pipeline Parallel	Regularity
raytracer	GRAMPS	Medium	99%	Yes	Irregular
hist-r/c	MapReduce	Small	97%	No	Irregular
lr-r/c	MapReduce	Small	99%	No	Regular
pca	MapReduce	Small	99%	No	Regular
mergesort	Cilk	Medium	99%	Yes	Irregular
fm	StreamIt	Large	43%	Yes	Regular
tde	StreamIt	Huge	8%	Yes	Regular
fft2	StreamIt	Medium	70%	Yes	Regular
serpent	StreamIt	Medium	86%	Yes	Regular
srad	CUDA	Small	99%	No	Regular
rg	CUDA	Small	99%	No	Regular

Table 6.2: Application characteristics. Pipeline parallelism denotes the existence of producer-consumer parallelism at the graph level.

6.6 Methodology

System: We perform all experiments on a 2-socket system with hexa-core 2.93 GHz Intel Xeon X5670 (Westmere) processors. With 2-way Simultaneous Multi-Threading (SMT), the system features a total of 12 cores and 24 hardware threads. The system has 256 KB per-core L2 caches, 12 MB per-socket L3 caches, and 48 GB of DDR3 1333 MHz memory (about 21 GB/s peak memory bandwidth). The processors communicate through a 6.4 GT/s QPI interconnect. This machine runs 64-bit GNU/Linux 2.6.35, with GCC 4.4.5. We ran all of our experiments several times and report average results; experiments were run until the 95% confidence intervals on each average became negligible ($< 1\%$).

Applications: In order to broadly exercise the schedulers, we have expanded the original set of GRAMPS applications [146] with a variety of examples from other programming models. Table 6.2 summarizes their qualitative characteristics, and Figure 6.3 shows a few representative graphs. The applications are:

- **raytracer** is the packetized ray tracer from [146]. We run it with no reflection bounces (ray-0), in which case it is a pipeline, and with one bounce (ray-1), in which case its graph has a cycle (as in Figure 6.1).

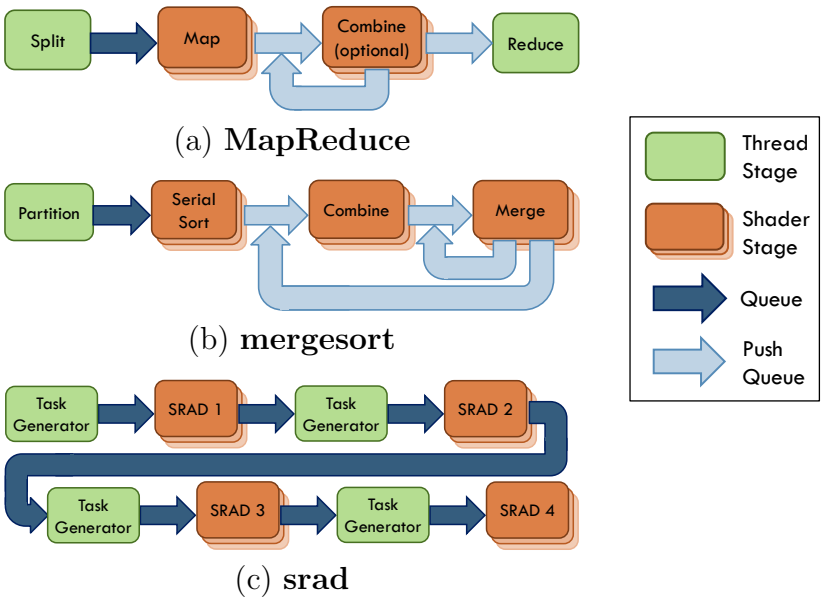


Figure 6.3: Example application graphs.

- **histogram, lr, pca** are MapReduce applications from the Phoenix suite [167]. We run **histogram** and **lr** in two forms: reduce-only (r) and with a combine stage (c).
- **mergesort** is a parallel mergesort implementation using Cilk-like spawn-sync parallelism. Its graph, shown in Figure 6.3, contains two nested loops.
- **fm, tde, fft2, serpent** are streaming benchmarks from the StreamIt suite [151]. All have significant pipeline parallelism and use ordered queues. **fm** and **tde** have very large graphs, with a small amount of data parallelism, while **fft2** and **serpent** have smaller graphs and more data parallelism.
- **srad, rg** are data-parallel CUDA applications. **srad** (Speckle Reducing Anisotropic Diffusion, an image-processing benchmark) was ported from the Rodinia suite [37], and **rg** (Recursive Gaussian) comes from the CUDA SDK [121].

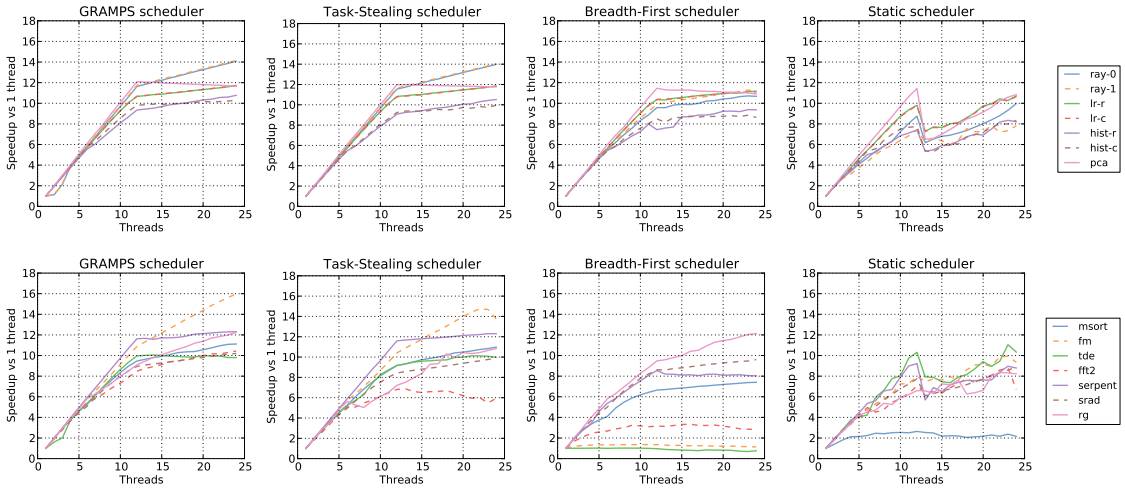


Figure 6.4: Scalability of GRAMPS, Task-Stealing, Breadth-First, and Static schedulers from 1 to 24 threads (12 cores).

6.7 Evaluation

6.7.1 GRAMPS Scheduler Performance

Figure 6.4 shows the speedups achieved by the GRAMPS scheduler from 1 to 24 threads. The knee that consistently occurs at 12 threads is where all the physical cores are used and the runtime starts using the second hardware thread per core (SMT). Overall, all applications scale well. With more cores, **srad** and **rg** scale sublinearly because they become increasingly bound by cache and memory bandwidth. They are designed for GPUs, which have significantly more bandwidth.

Figure 6.5 gives further insight into these results. It shows the execution time breakdown of each application when using all 24 hardware threads. In this section, we focus on the results with the GRAMPS scheduler (the leftmost bar for each app). Each bar is split into four categories, showing the fraction of time spent in application code, scheduler code, buffer manager, and stalled (which in the GRAMPS scheduler means stealing, with no work to execute). This breakdown is obtained with low-overhead profiling code that uses the CPU timestamp counter, which adds $\leq 2\%$ to the execution time.

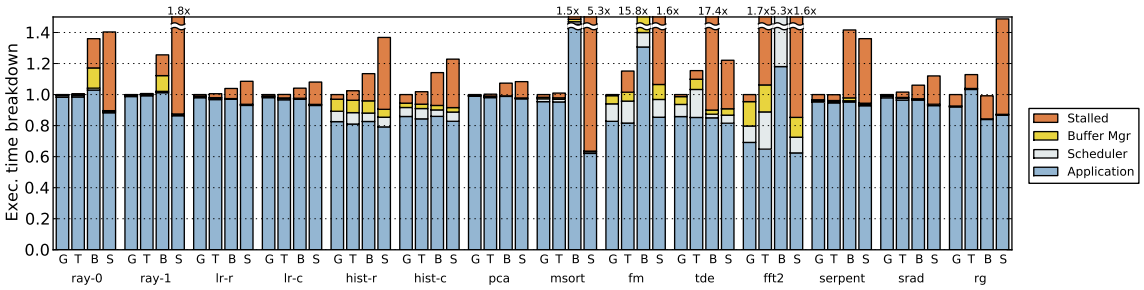


Figure 6.5: Runtime breakdown of GRAMPS (G), Task-Stealing (T), Breadth-First (B), and Static schedulers (S) at 24 threads. Each bar shows the execution time breakdown. Results are normalized to the runtime with the GRAMPS scheduler.

Overall, results show that GRAMPS is effective at finding and dynamically distributing parallelism: applications spend minimal time without work to do. Furthermore, runtime overheads are small: the majority of workloads spend less than 2% of the time in the scheduler and buffer manager. Even tracking the tens of stages in **fm** takes only 13% of elapsed time in scheduling overheads. The worst-case buffer manager overhead happens in queue-intensive **fft2**, at 15%.

Finally, Table 6.3 shows the average and maximum footprints of the GRAMPS scheduler. Footprints are reasonable, and always below the maximum queue sizes, due to backpressure providing strict footprint bounds.

6.7.2 Comparison of Scheduler Alternatives

We now evaluate the differences among the scheduling alternatives discussed in Section 6.2. Figure 6.4, Figure 6.5, and Table 6.3 show the scalability, execution time breakdowns, and footprints for the different schedulers, respectively.

Task-Stealing: For applications with simple graphs, the Task-Stealing scheduler achieves performance and footprint results similar to the GRAMPS scheduler. However, it *struggles on applications with complex graphs*: **fm** and **tde**, and applications with ordering, **fft2**.

fm and **tde** have the most complex graphs, with 121 and 412 stages respectively (Table 6.2), and have abundant pipeline parallelism but little data parallelism, and

App	GRAMPS		Task-Stealing		Breadth-First		Static	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
ray-0	215.9 KB	287.8 KB	191.7 KB	276.5 KB	23096 KB	51648 KB	322.7 KB	722.7 KB
ray-1	361.1 KB	447.4 KB	327.5 KB	424.6 KB	31257 KB	78706 KB	1133 KB	2271 KB
lr-r	22.9 KB	40.7 KB	22.9 KB	40.9 KB	24.0 KB	44.0 KB	23.7 KB	43.8 KB
lr-c	8.1 KB	8.8 KB	7.9 KB	8.2 KB	15.2 KB	30.2 KB	9.8 KB	12.8 KB
hist-r	4922 KB	9658 KB	4925 KB	9654 KB	4695 KB	9737 KB	4903 KB	9653 KB
hist-c	1860 KB	3097 KB	1864 KB	3096 KB	1504 KB	3093 KB	1855 KB	3099 KB
pca	0.7 KB	1.3 KB	0.4 KB	0.4 KB	89.2 KB	178.8 KB	4.6 KB	8.0 KB
msort	2.0 KB	7.0 KB	1.4 KB	4.6 KB	6.1 KB	10.4 KB	5.3 KB	18.0 KB
fm	1672 KB	4277 KB	2245 KB	3691 KB	165145 KB	563173 KB	29646 KB	57391 KB
tde	3989 KB	6662 KB	18399 KB	36231 KB	89843 KB	179282 KB	18379 KB	31071 KB
fft2	261.3 KB	376.0 KB	149.8 KB	211.0 KB	75624 KB	80096 KB	1184 KB	1395 KB
serpent	79.1 KB	88.0 KB	68.7 KB	73.2 KB	1031 KB	1048 KB	735.8 KB	1003 KB
srad	0.9 KB	1.6 KB	0.4 KB	0.5 KB	40.0 KB	80.0 KB	2.6 KB	8.2 KB
rg	0.7 KB	1.4 KB	0.4 KB	0.5 KB	1.9 KB	5.0 KB	0.6 KB	1.6 KB

Table 6.3: Average and maximum footprints of different schedulers.

ordered queues. In both **fm** and **tde**, Task-Stealing is unable to keep the system fully utilized, as evidenced by the larger Stalled application times, due to the simple LIFO policy and lack of backpressure, which also cause larger footprints and overheads. In **fft2**, scalability is limited by the last stage in the pipeline, a Thread consumer. Since this workload has ordering requirements, the LIFO task ordering causes significant head-of-line blocking; packets are released in bursts, which causes this stage to bottleneck sooner. This bottleneck shows as the large Stalled part of the execution breakdown in **fft2** under Task-Stealing. In contrast, with graph knowledge, GRAMPS uses FIFO task queuing on ordered stages (Section 6.4). Hence packets arrive almost ordered, and the receiving stage does not bottleneck.

Breadth-First: The Breadth-First scheduler cannot take advantage of pipeline parallelism. Consequently, it *matches the GRAMPS scheduler only on those applications without pipeline parallelism*, **srad** and **rg**. In other applications, the one-stage-at-a-time approach significantly affects performance and footprint.

Performance is most affected in **fm** and **tde**, which are highly pipeline-parallel but not data-parallel (Table 6.2). Looking at the execution breakdown for those two

applications, we see that Breadth-First scheduling leaves the system starved for work for up to 95% of the time. Compared to GRAMPS, the slowdowns are as high as 15.8x and 17.4x, respectively.

In terms of footprint, the large amount of intermediate results generated by each stage put high pressure on the memory system and the buffer manager. Footprint differences are most pronounced in **raytracer** and the StreamIt applications (Table 6.3). For example, the **raytracer**'s worst-case footprint is 447 KB with GRAMPS, but 78.7 MB with Breadth-First. This turns into a larger buffer manager overhead, which takes 12% of the execution time (Figure 6.5). Larger footprint reduces the effectiveness of caches, hurting locality, as seen from the slightly higher application time.

Static: The Static scheduler trades off load balancing for near-optimal static work division and minimized producer-consumer communication. Although this is likely a good trade-off in embedded/streaming systems with fully static applications, we see that *it is a poor choice when either the system or the application is dynamic.*

Focusing on the scalability graph (Figure 6.4), we see that from 1 to 12 threads the static scheduler achieves reasonable speedups for highly regular applications: **pca**, **lr**, and **tde** get close to linear scaling. However, more irregular applications experience milder speedups, e.g., up to 7x for the **raytracer**. The worst-performing application is **mergesort**, which has highly irregular packet rates, and static partitioning fails to generate an efficient schedule.

As we move from 12 to 13 threads, performance drops in all applications. At this point, SMT starts being used in some cores, so threads run at different speeds. While the other schedulers easily handle this by performing load balancing, this fine-grain variability significantly hinders the Static scheduler. Interestingly, the execution time breakdown (Figure 6.5) shows that the static scheduler actually achieves lower application times, due to optimized locality. However, these improvements are more than negated by load imbalance, which increases the time spent waiting for work.

In summary, we see that GRAMPS and Task-Stealing achieve the best performance overall. However, Task-Stealing's simple LIFO queuing does not work well for

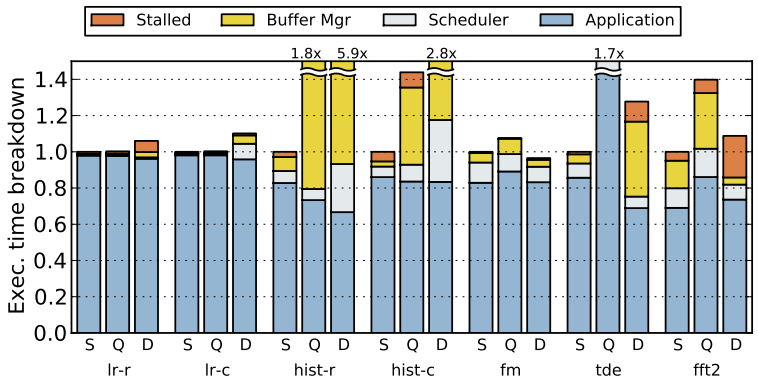


Figure 6.6: Runtime breakdown with GRAMPS scheduler on queue-heavy applications, using the three alternative buffer managers: packet-stealing (S), per-queue (Q), and dynamic memory (D).

complex application graphs or graphs with ordering, and cannot bound footprint due to lack of backpressure. While Breadth-First scheduling takes advantage of data parallelism, it cannot extract pipeline parallelism. Breadth-First scheduling also cannot take advantage of producer-consumer communication typical of pipeline-parallel programs, causing memory footprint to be extremely large. Static scheduling is able to effectively schedule for locality, but cannot handle irregular applications or run-time variability in the underlying hardware. More importantly, we observe that GRAMPS' dynamic scheduling overheads are *mostly negligible* and locality is not significantly worse than what is achieved by locality-optimized Static schedules. This shows that, contrary to conventional wisdom, dynamic schedulers can efficiently schedule complex pipeline-parallel applications.

6.7.3 Comparison of Buffer Management Strategies

All the results previously shown have used the proposed packet-stealing buffer manager. We now evaluate the importance of this choice. Figure 6.6 compares the performance of the different buffer management approaches discussed in Section 6.4, focusing on applications where the choice of buffer manager had an impact.

We observe that the packet-stealing approach achieves small overheads and good locality. In contrast, the dynamic buffer manager often has significant slowdowns, as

frequent calls to malloc/free bottleneck at the memory allocator. We used tcmalloc as the memory allocator [63], which was the highest-performing allocator of those we tried (ptmalloc2, hoard, dlmalloc, and nedmalloc). Compared to the stealing buffer manager, the worst-case slowdown, 5.9x, happens in **histogram**, which is especially footprint-intensive.

The per-queue buffer manager shows lower overheads than the dynamic scheme. However, overheads can still be large (up to 80% in **histogram**), and more importantly, having per-queue slabs can significantly affect locality, as seen by the higher application times in several benchmarks. This is most obvious in **tde**, where the large number of global per-queue pools (due to the large number of stages) causes application time to increase by 50%. In contrast, the stealing allocator uses a reduced number of thread-local LIFO pools, achieving much better locality.

Overall, we conclude that buffer management is an important issue in GRAMPS applications, and the stealing buffer manager is a good match to GRAMPS in systems with cache hierarchies, achieving low overheads while maintaining locality.

6.8 Additional Related Work

We now discuss additional related work not covered in Section 6.2.

Although many variants of Task-Stealing schedulers have been proposed, one of the most relevant aspects is whether the scheduler follows a work-first policy (moving depth-first as quickly as possible) or a help-first policy (producing several child tasks at once). Prior work has shown there are large differences between these approaches [67], and has proposed an adaptive work-first/help-first scheduler [68]. In GRAMPS, limiting the number of output packets produced by a Thread before it is preempted controls this policy. Our Task-Stealing scheduler follows the work-first policy, but we also tried the help-first policy, which did not generally improve performance, as the benefits of somewhat reduced preemption overheads were countered by higher memory footprints. The GRAMPS scheduler follows the help-first policy, preempting threads after several output packets, and leverages backpressure to keep footprint limited.

Some Task-Stealing models implement limited support for pipeline parallelism; TBB [81] supports simple 1:1 pipelines, and Navarro et al. [118] use this feature to study pipeline applications on CMPs, and provide an analytical model for pipeline parallelism. As we have shown, Task-Stealing alone does not work well for complex graphs.

Others have also observed that Breadth-First schedulers are poorly suited to pipeline-parallel applications. Horn et al. [78] find that a raytracing pipeline can benefit by bypassing the programming model and writing a single uber-kernel with dynamic branches. Tzeng et al. [154] also go against the programming model by using uber-kernels, and implement several load balancing strategies on GPUs to parallelize irregular pipelines. They find that task-stealing provides the least contention and highest performance. The techniques presented in this paper could be used to extend GRAMPS to GPUs.

Although most work in streaming scheduling is static, prior work has introduced some degree of coarse-grain dynamism. To handle irregular workloads, Chen et al. propose to pre-generate multiple schedules for possible input datasets and steady states, and switch schedules periodically [38]. Flextream [77] proposes online adaptation of offline-generated schedules for coarse-grain load balancing. These techniques could fix some of the maladies shown by Static scheduling on GRAMPS (e.g., SMT effects). However, applications with fine-grain irregularities, like **raytracer** or **mergesort**, would not benefit from this. Feedback-directed pipelining [148] proposes a coarse-grain hill-climbing partitioning strategy to maximize parallelism and power efficiency on pipelined loop-parallel code. While the power-saving techniques proposed could be used by GRAMPS, its coarse-grained nature faces similar limitations.

Finally, we have focused on parallel programming models commonly used in either general-purpose multi-cores, GPUs and streaming architectures. We have not covered programming models that target clusters, MPPs or datacenter-scale deployments. These usually expose a multi-level memory organization to the programmer, who often has to manage memory and locality explicitly. Examples include MPI [143], PGAS-based models such as UPC [30] or Titanium [166], and more recent efforts like Sequoia [58].

6.9 Summary

In this chapter, we have presented a scheduler for pipeline-parallel programs that performs fine-grain dynamic load balancing efficiently. Specifically, we implement the first real runtime for GRAMPS, a recently proposed programming model that focuses on supporting irregular pipeline-parallel applications. Our evaluation shows that the GRAMPS runtime achieves good scalability and low overheads on a 12-core, 24-thread machine, and that our scheduling and buffer management policies efficiently schedule simple and complex application graphs while preserving locality and bounded footprint.

Our scheduler comparison indicates that both Breadth-First and Static scheduling approaches are not broadly suitable on general-purpose processors, and that GRAMPS and Task-Stealing behave similarly for simple application graphs. However, as graphs become more complex, GRAMPS shows an advantage in memory footprint and execution time because it is able to exploit knowledge of the application graph, which is unavailable to Task-Stealing.

Chapter 7

ADM: Flexible Architectural Support for Fine-Grain Scheduling

7.1 Introduction

As we have seen in Chapter 6, dynamic scheduling has significant advantages over static scheduling. However, as the number of cores scales up, these schedulers need finer-grain tasks to expose enough parallelism. Unfortunately, as we discussed in Section 2.3.2, managing tasks of a few thousand instructions is particularly challenging, as the runtime must ensure load balance without compromising locality and introducing small overheads. Software-only schedulers can implement various scheduling algorithms that are tailored to specific programming models or even applications, but suffer significant overheads as they synchronize and communicate task information over the deep cache hierarchy of a large-scale CMP. To reduce these costs, hardware-only schedulers like Carbon [101], which implement task queuing and scheduling in hardware, have been proposed. However, a hardware-only solution fixes the scheduling algorithm and leaves no room for other uses of the custom hardware.

In this chapter we present a combined hardware-software approach to build fine-grain schedulers that retain the flexibility of software schedulers while being as fast and scalable as hardware ones. We introduce *asynchronous direct messages (ADM)*, a

general-purpose hardware primitive that supports sending and asynchronously receiving short messages between cores at low overhead without additional synchronization or going through the coherence protocol. ADM is tailored to integrate with cache-coherent CMPs and the shared-memory programming models for such systems, and is inspired by previous efforts in integrating message-passing in distributed shared memory machines [1, 111]. ADM provides *user-level support* for relatively infrequent messages for control purposes, while data accesses and communication occur as usual through the cache hierarchy. ADM is sufficient to implement novel, software-mostly fine-grain schedulers that rely on low-overhead messaging to efficiently coordinate scheduling and transfer task information. Since software determines the scheduling algorithm, we can easily tailor it to the programming model or application. For example, we can adjust the stealing policy to improve locality, track dependencies between tasks, implement fast reduction and barrier operations, and use hierarchical scheduling approaches that scale better with the size of the CMP. Such optimizations allow ADM-based schedulers to match or outperform hardware-only approaches like Carbon, sometimes by by large margins, despite using simpler hardware. This chapter presents the following contributions:

1. We introduce ADM, a simple but general hardware mechanism to send messages between cores. ADM allows user-level code to send short messages (0-6 words) directly from registers, and to receive them, either synchronously or asynchronously via a user-level interrupt handler. The hardware is virtualizable, preserves message ordering, provides guaranteed delivery, and is independent of the cache hierarchy.
2. We develop and present a set of novel Task-Stealing schedulers that leverage the messaging hardware to manage fine-grain parallelism. Specifically, we use a subset of worker threads to coordinate stealing in a distributed and scalable fashion. ADM allows threads to maintain task queues in thread-local storage, even when stealing occurs, and to overlap communication with useful computation. Although in this chapter we focus on Task-Stealing schedulers, ADM can also accelerate other schedulers that use stealing as a load-balancing mechanism, such as our GRAMPS runtime presented in Chapter 6.
3. We evaluate ADM for multithreaded CMPs with up to 128 cores (256 threads)

using a set of challenging fine-grain applications. We find that our approach clearly outperforms software-only schedulers by up to a factor of 3.8× and matches or exceeds the performance of Carbon. When we tailor the ADM scheduler to the application, it outperforms Carbon by up to 70%.

7.2 Background and Motivation

7.2.1 Current Scheduling Approaches

To implement fine-grain schedulers on a cache-coherent CMP, we can either use a software-only solution in which threads communicate through shared memory, or leverage special-purpose hardware. In large-scale CMPs, both approaches have serious disadvantages.

Software-only schedulers maintain queues in software, and threads communicate and exchange work implicitly through shared memory. Several optimized algorithms have been proposed to avoid the use of locks in most local enqueue/dequeue operations [60] or to use non-blocking stealing protocols [10, 35]. Still, stealers need to perform multiple remote cache accesses to find and obtain new work, which will take hundreds of cycles through the cache hierarchy of a large-scale CMP. If stealing is infrequent or tasks are large, stealing overheads can be amortized. However, irregular or fine-grain workloads with frequent steals suffer from large penalties even with the most optimized protocols, due to memory latency, synchronization, or contention. These overheads will only become worse as we increase the number of cores on a CMP, because 1) the latency of a remote or a shared cache access increases, and 2) the amount of work per thread decreases, leading to shorter phases and more frequent steals.

Hardware-only schedulers, such as Carbon [101], introduce specialized hardware that handles all aspects of work-stealing. Carbon uses a centralized *global task unit (GTU)*, which contains one hardware LIFO queue per thread. Software uses special instructions to enqueue and dequeue task descriptors directly to/from registers. Task descriptors have a fixed size of 4 64-bit words. A small *local task unit*

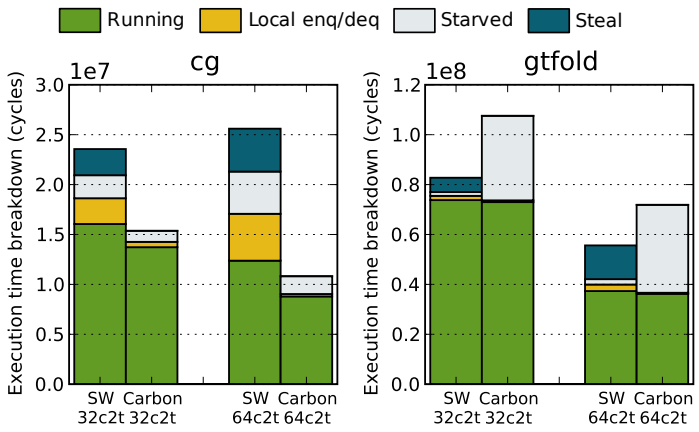


Figure 7.1: Execution time of cg and gtfold using Carbon and software schedulers, on CMPs with 32 and 64 dual-threaded cores (for a total of 64 and 128 threads).

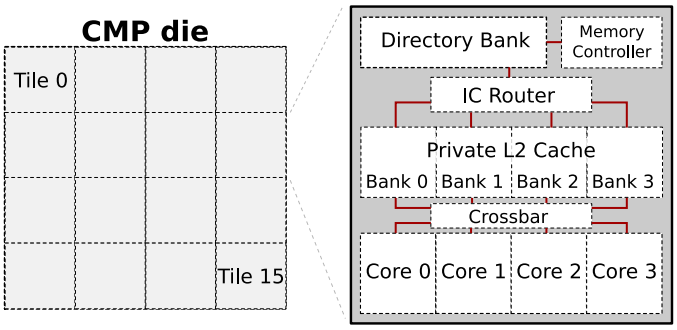
(LTU) per core is used as a task buffer to hide enqueue and dequeue latencies from the GTU. Work-stealing is implemented in hardware in the GTU by moving tasks between queues. Since the queues in the GTU are bounded, the runtime system cannot rely exclusively on Carbon for task buffering. Each worker thread maintains an unbounded task queue in software, where it can enqueue and dequeue new tasks locally. A portion of these tasks are enqueued in the corresponding hardware queue in the GTU to allow for work-stealing. Trying to dequeue a task when the GTU is empty blocks the thread. When all threads are blocked, the GTU sends a special task to every thread to signal the end of the parallel phase. The GTU generates a user-level interrupt when the capacity of a hardware queue reaches an upper or lower threshold to allow each thread to overflow to or refill tasks from the software queue.

Carbon addresses the performance issues of software-only scheduling, but hard-wires the scheduling policies, such as the structure of queues and the order or granularity of stealing. Hence, it is difficult to use with applications or programming models that require alternative algorithms. Figure 7.1 illustrates this issue. It shows the execution time breakdown between software-only scheduling and Carbon (see Section 7.5 for the experimental methodology), for two applications. The first one, cg, is fine-grain, irregular, and has short phases. Carbon can do load-balancing very

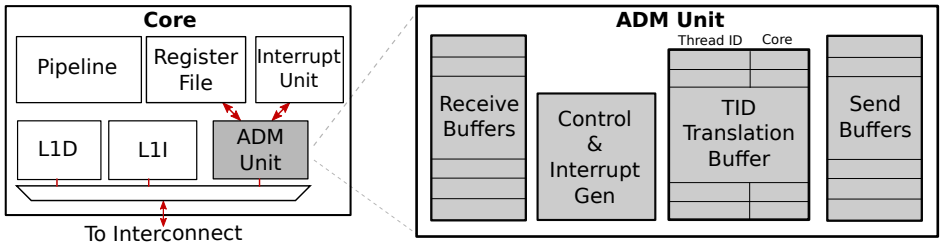
efficiently, thus outperforming the software scheduler by 2.2× at 128 threads. This reduction comes both from eliminating the stealing and enqueuing/dequeuing overheads of software, and performing better balancing (as threads spend less time starved, i.e., without work to execute). The second one, gtfold, is also fine-grain but can benefit from a FIFO scheduling algorithm (details in Section 7.6). The software FIFO scheduler, as shown, outperforms Carbon by 40% at 128 threads by reducing the starvation time. While it is possible to extend Carbon to capture a few scheduling variations, implementing all possible scheduling algorithms in hardware can be prohibitively expensive.

7.2.2 Fast and Flexible Fine-Grain Scheduling

We strive to find a balance between speed and flexibility for fine-grain scheduling. We also want to minimize the required hardware by introducing simple primitives that have multiple uses rather than fixed-function hardware. We note that Carbon can be deconstructed in three elements: hardware task queues, logic for stealing across these queues, and messaging hardware for fast communication of tasks. As we will see, implementing queues and policies in software is not much slower than using hardware, as long as all the scheduler state is kept *thread-local*, avoiding remote memory accesses. To allow this, we advocate keeping a fast messaging component in hardware and exposing it directly as a general-purpose mechanism. This avoids the high penalties of communicating and synchronizing through the memory hierarchy when task stealing or coordination among software schedulers is needed, and allows to overlap communication with useful computation. Moreover, scheduling in software with fast messaging for communicating control information allows us to create runtime systems tailored to the characteristics and requirements of specific applications or programming models.



(a) Target CMP, shown in a 64-core configuration with 16 tiles.



(b) The microarchitecture of the ADM unit added to each core.

Figure 7.2: Target CMP architecture and modifications needed to implement ADM.

7.3 Asynchronous Direct Messages

We now describe Asynchronous Direct Messages (ADM), the flexible messaging mechanism that we propose to accelerate fine-grain schedulers. ADM adds an extra messaging unit per core that works with any cache hierarchy or coherence protocol. To focus the discussion, we consider cache-coherent, tiled CMPs with a packet-switched interconnect, as the one shown in Figure 7.2a.

To provide low-overhead messaging with small payloads, messages are sent and received directly through registers instead of using memory-mapped buffers. Software threads initiate a send with a single instruction. Reception can be synchronous, using a receive instruction, or asynchronous, via a user-level message handler. To enforce atomicity of accesses to data structures shared by handler and non-handler code (e.g., the task queue), messaging interrupts can be disabled or enabled by non-handler code. The microarchitecture has limited message buffers, which are backed by software

buffering if needed. Normal operation happens fully at user-level, but the architecture is virtualizable, and minimal OS support is required. The system preserves message ordering between sender and receiver and guarantees message delivery because these features greatly simplify writing ADM-based schedulers.

7.3.1 Microarchitecture and ISA

We modify a baseline CMP in two ways. First, we add an ADM unit to each core, shown in Figure 7.2b. This unit buffers received messages, translates thread IDs to physical cores on message sending, and interfaces to the register file (to transfer message payloads), the core's interrupt unit (to deliver message reception interrupts), and the interconnect. Second, we use an extra virtual network [46, 47] in the packet-switched interconnect to route message packets. This only requires a moderate amount of extra buffering in the routers (e.g., 9KB of SRAM space in a 64-node CMP), and avoids deadlocks due to interference with the coherence protocol traffic.

The ADM unit includes one receive buffer per hardware thread context, implemented as a small circular buffer using SRAM memory. Thread ID to core translation is performed by the *Thread ID Translation Buffer (TTB)*, a small associative memory that caches `(TID, core)` pairs. The TTB is software-managed; if it cannot translate a destination's thread ID when sending, it triggers a privileged interrupt handler that refills it with the appropriate translation. Our hierarchical runtimes do not use all-to-all communication, but instead each thread communicates with a fixed-size subset of the threads. Thus, we only require small receive buffers (that can hold around 16 4-word messages) and TTBs (of 16∼32 entries) for full performance. Furthermore, these per-core structures do not need to grow with the size of the CMP.

Table 7.1 summarizes the hardware-software interface, including the extra instructions to send/receive messages and the new interrupts and exceptions introduced by ADM. The send instruction is blocking, and the instruction is considered completed when the message is copied to the send buffer. Each message is transmitted in the interconnect in a single packet, but using multiple flits [46, 47].

Instruction	Description
`adm_send r1, r2`	Sends message of (r1) words to thread with ID (r2). The payload can have 0–6 words, taken from registers %o0-%o5
`adm_peek r1, r2`	Returns the source and message length at the head of the receive buffer, or a -1 length if the buffer is empty.
`adm_rx r1, r2`	Returns the source and message length at the head of the receive buffer, and writes its payload to registers %o0-%o5. Blocks on an empty buffer.
`adm_ei/di`	Enables or disables ADM receive handler interrupts.

Event	Type	Privileged
Receive	Interrupt	No
Receive buffer under/overflow	Interrupt	Yes
TTB miss	Exception	Yes
TTB remote invalidate	Interrupt	Yes

Table 7.1: ISA extensions and new events introduced by ADM, assuming a SPARCv9 ISA.

7.3.2 Guaranteed Delivery and Ordering

To simplify software schedulers, our hardware design preserves message ordering for each source-destination pair and provides guaranteed delivery. We first focus on how to implement guaranteed delivery. Suppose there are no send buffers in the ADM units. This implies that neither the interconnect nor the receiving buffer can drop messages and that all messages should eventually be dequeued by the receiving core. These requirements can lead to deadlock scenarios. For example, if two threads send messages to each other and neither is dequeuing them, the receive buffers will become full as well as any interconnect buffers between the two cores. Further send attempts will be blocked and the two threads will deadlock. To guarantee the absence of deadlock, we must ensure that threads will eventually dequeue any message they receive. One option is to prohibit reception handlers from blocking (e.g., by disallowing sending messages or acquiring locks) [156], but this is too restrictive for software. Instead, we include a second, privileged interrupt handler, which is triggered when the thread's receive buffer becomes full and dequeues half of the messages from the back of the buffer. To preserve ordering, the privileged handler marks the

first non-dequeued message. When that message is dequeued by the thread, the privileged handler triggers again, refilling the buffer with the dequeued messages. Using the privileged handler, we provide unbounded receive buffers in software.

We structured our runtimes to almost never exceed the size of the receive buffers, avoiding the performance penalty of the second interrupt handler. However, bad-behaving user-level software (e.g., a buggy or malicious program) may cause serious interference with other programs that use ADM by quickly filling up the virtual network buffers with messages before the privileged handler can free space in the receive buffer. To avoid this, we have a send buffer per thread and use a simple ACK/NACK flow control scheme. Note that although this avoids clogging the network with messages, the privileged interrupt handler is still needed to prevent deadlock. Small send buffers suffice, since ADM is not continuously used. In our runtimes, send buffers that hold 16 messages are sufficient to cover message bursts. Send buffers also make it easier to preserve message ordering under virtualization (Section 7.3.3).

The flow control protocol between sender and receiver must preserve message ordering between each source-destination pair. In general, we can either use deterministic routing in the interconnect or implement a flow control protocol with reordering at the endpoints. We opt for deterministic routing in our evaluation, since networks in cache-coherent CMPs are often lightly loaded. Since the interconnect does not reorder messages, all we need to ensure is that, when the receiver R issues a NACK to sender S, all the messages in flight from S to R are discarded. To do this, every receiver keeps one bit per sender $B_{R \leftarrow S}$, and every sender keeps one bit per receiver $B_{S \rightarrow R}$. All the bits are initially 0. S stamps all its messages to R with $B_{S \rightarrow R}$. When R NACKs a message, it flips $B_{R \leftarrow S}$. When S receives the NACK, it flips $B_{S \rightarrow R}$ and tries retransmission of the messages to the receiver. Finally, R ignores all packets arriving from S with bit stamp not matching $B_{R \leftarrow S}$.

7.3.3 Virtualization

The OS needs to be aware of ADM, and perform some extra tasks to interact with it. First, it needs to assign a unique thread ID to each thread in the system, and

maintain a mapping between thread IDs and physical cores for scheduled threads that the TTB refill handler can use. Since the TTB is software-managed, the OS can decide which thread pairs can communicate with ADM. For example, threads from multiple processes could be allowed to communicate to implement fast user-level IPC. Second, to support thread migration and descheduling, we introduce a lazy TTB invalidation mechanism. When a core receives a message intended for another thread context, it sends a NACK back to the sender, indicating that the TTB entry is stale. This NACK triggers a privileged interrupt in the sender, which either refills the TTB with the correct mapping and resends the messages for this destination in the send buffer if the thread was migrated, or saves them in a software queue and invalidates the stale TTB entry if the thread is switched out. Finally, the OS should ensure that a thread's send buffer is empty before migrating it to avoid losing message ordering under migrations.

7.4 Runtime Systems

We now present our baseline software-only runtime and the runtimes that utilize the hardware features of Carbon and ADM. For this study, we focus on Task-Stealing runtimes with either a task-parallel or a loop-parallel API. In all cases, the application code is the same. The application programmer writes code for a shared-memory CMP and is oblivious of the use of Carbon or ADM. Only the low-level runtime code interacts with additional hardware features.

7.4.1 Task-Parallel Runtimes

API: All the task-parallel runtimes implement the same simple interface, consisting of two functions:

- `void enqueue(Task t)`: Enqueues the task identifier `t` for execution in the current parallel phase. Task identifiers consist of four 64-bit words.
- `bool dequeue(Task& t)`: Tries to dequeue a task identifier from the current parallel phase. If successful, returns `true` and copies the task to `t`. Otherwise,

returns `false`, signaling the end of the parallel phase.

Software-only runtime: Our baseline runtime is a highly optimized work-stealing scheduler. It uses Chase-Lev circular work-stealing deques [35], which require an atomic operation per steal, but not in local enqueues or most local dequeues. Local enqueues and dequeues are done in LIFO order. The stealing protocol is fully non-blocking: local enqueues/dequeues and steals to the same queue can happen concurrently, and a stealing thread never blocks waiting on other stealer to finish. The phase termination protocol is as in the X10 work-stealing scheduler [44]. We carefully control memory layout to avoid false sharing and maximize spatial locality.

We explored tuning the stealing policy in two dimensions: victim selection (random, round robin or nearest neighbor) and tasks to grab per steal (one or half of the queue). In our experiments, we find that trying to steal from nearest neighbors first outperforms random or round-robin stealing due to improved locality, and stealing half of the victim's queue is preferable to stealing one task to amortize software overheads and preserve inter-task locality. Therefore, the software runtime uses these policies in the evaluation.

Carbon runtime: The Carbon runtime operates as outlined in Section 7.2. Each worker has a private and unbounded LIFO task queue in software that is used when its hardware queue fills up. Work-stealing is enabled by maintaining at least a portion of the task queues in the hardware LIFO queues. The GTU hardware performs work-stealing in the background, and the LTUs fetch and prefetch tasks from the GTU. When the GTU needs to send a task to the LTU of a specific thread, it first attempts a dequeue from its corresponding hardware queue. If the queue is empty, it steals from a non-empty victim queue in a single cycle. As long as there are tasks in the hardware queues, the latencies of work-stealing and the communication between the GTU and the LTUs are hidden from the worker threads. Since the GTU cannot reclaim tasks from the LTUs, it does not serve prefetches when there are few tasks in the GTU to avoid load imbalance [101]. We optimized the runtime to minimize overflows and underflows of the hardware queues. As with the software runtime, we have explored different stealing policies in the GTU, and choose to steal half of the tasks from the

Name	Type	Description	Format
UPDATE	↑	Inform manager of task count	<level, numTasks>
STEAL	↓	Notify victim to perform a steal	<level, tasksToXfer, stealer>
TASK	→	Transfer one task to stealer	<taskDescriptor, isLastTask>
VICTIM_UPDATE	↑	Victim notifies manager of steal outcome & new task count	<level, tasksXferd, tasksLeft>
STEALER_UPDATE	↑	Stealer notifies manager that steal is over & new task count	<level, numTasks>
UNBLOCK	↓	Notify end of phase	<level>

Table 7.2: Messages in the ADM runtime protocols. The type column indicates how the message flows in the hierarchy of workers and managers: down (↓), up (↑), or between workers (→). The level field indicates the tree level of the manager in the hierarchical runtime, and is not used in the centralized runtime.

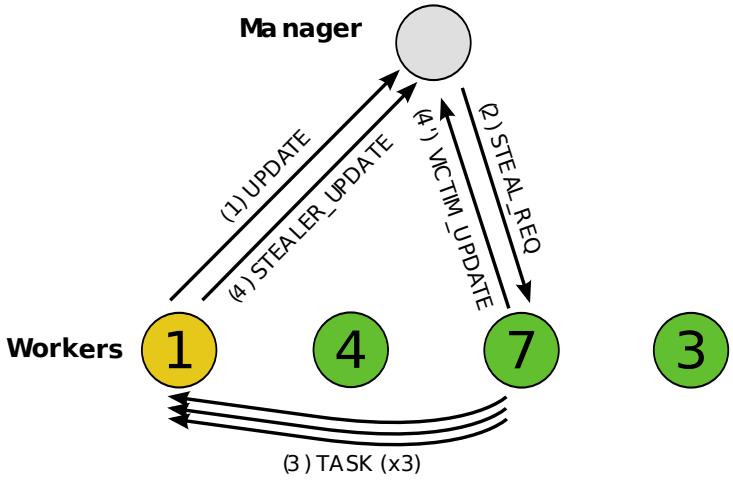


Figure 7.3: Messages involved in a steal in the centralized ADM runtime. The number in each worker indicates the tasks in its queue.

nearest-neighbor queue, both for performance reasons and for consistency with the software runtime policy.

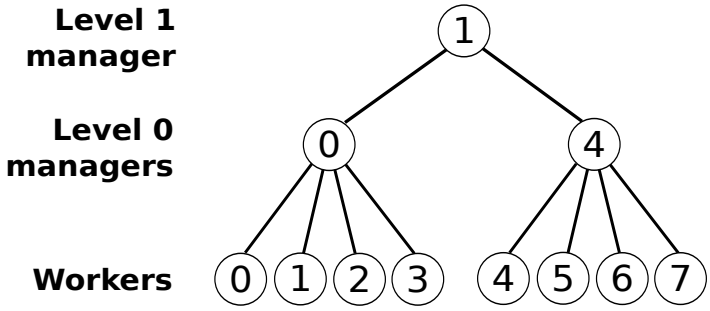
ADM runtimes: In ADM runtimes, threads can adopt the roles of *workers* or *managers*. A worker executes the program, enqueuing and dequeuing tasks in thread-local software queues. A manager handles task distribution, load balancing, and parallel phase termination by exchanging messages with workers. Managers do not maintain task queues themselves. A thread can act as a dedicated worker, as a

dedicated manager, or as both worker and manager, switching between the two roles as messages arrive and the interrupt handler is triggered.

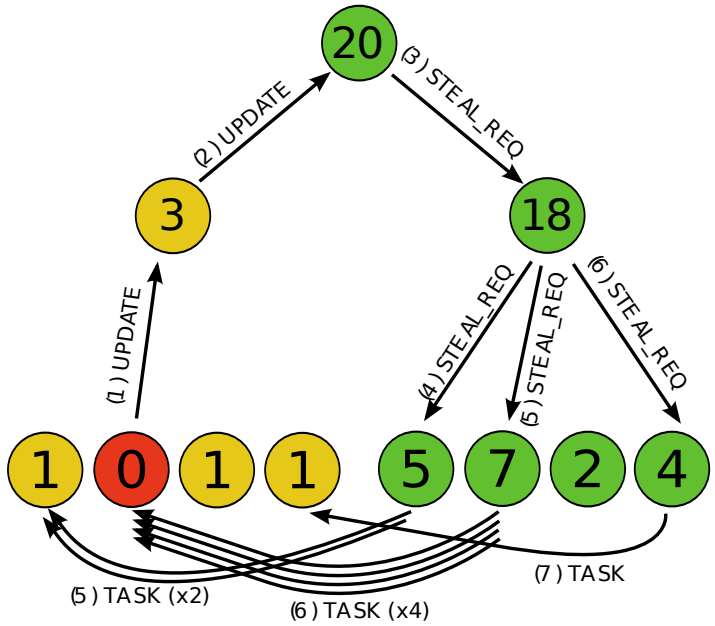
A simple ADM-based runtime can be *centralized*, using a single manager to coordinate all the threads, or *distributed* if it uses multiple managers. Our centralized runtime operates with four kinds of messages: updates, steal requests, task transfers, and unblocks. The details of each message are shown in Table 7.2. Workers send *update* messages to notify the manager about changes in the number of locally queued tasks, using the `adm_send` instruction in the functions for task enqueuing and dequeuing. To avoid saturating the manager, workers send updates only when their queues exceed exponentially varying thresholds. When an update message arrives, the interrupt handler invokes the manager code. The manager initiates and coordinates steals among workers based on its approximate knowledge of the number of tasks in each worker, as shown in Figure 7.3. If the number of tasks of a worker S (stealer) goes below an *underflow threshold*, the manager sends a *steal request* message to notify the worker V (victim) with the most tasks that it should send a portion of its queue to worker S. Worker V sends tasks to S, using one *task* message per task descriptor. When the steal is finished, S and V send updates to the manager. These updates may trigger further rebalances. A worker that tries to dequeue from an empty queue blocks and sends an update to its manager. When all the workers block, the manager sends an *unblock* message to every worker to signal the end of the parallel phase.

The centralized runtime is simple, but does not scale to large thread counts (e.g., 64 or 128), even when using a dedicated thread for the manager. If frequent stealing is needed, the manager quickly saturates when matching stealers and victims. Additionally, if the program has short phases, the single manager takes a long time to detect and signal phase termination.

To improve scalability, we implemented a *hierarchical* ADM-based runtime, with multiple levels of managers organized in a tree, as shown in Figure 7.4a. A level-0 manager directly coordinates a sub-group of workers, while managers at higher levels coordinate groups of managers down below. The hierarchical scheduler uses the same set of messages as in the centralized implementation, detailed in Table 7.2: updates flow up the tree, steal requests and unblocks flow down the tree, and task



(a) Workers and managers



(b) Level 1 steal

Figure 7.4: Organization and operation of the hierarchical ADM runtime. An 8-thread, radix-4 runtime is shown. (a) indicates which threads perform the roles of workers and managers. Note how the manager threads are also workers. (b) shows the messages involved in a multi-level steal. In (b), the numbers indicate tasks left in each worker, or task aggregates in managers. The update messages sent after the steal are omitted for clarity.

transfers happen between workers. Each manager keeps an approximate count of the aggregate number of tasks in its child partitions, and initiates steals within its partition when needed to counter imbalance. Hence, there can be steals spanning

multiple levels, as Figure 7.4b shows. In these multi-level steals, managers distribute the steal request among its children, so a single steal request can rebalance two whole partitions. The i^{th} worker of the victim partition always transfers tasks to the i^{th} stealer worker. This may require additional rebalances in the stealer partition, but enables managers to keep only aggregate task counts. This implementation provides global load balancing, but at the same time improves locality by first solving local imbalances with steals from nearby threads. Compared to a centralized runtime, this approach amplifies stealing bandwidth, allowing for frequent steals. We observe that radix-4 to radix-16 configurations perform best for our workloads, with marginal performance differences between them. Larger radices exhibit reduced performance due to manager saturation. Our evaluation uses a radix-8 tree.

Finally, the scheduler needs to be aware of the limitations of ADM. As explained in Section 7.3, ADM operates very efficiently as long as the TTB and receive buffer capacities are not exceeded. TTB overflow is avoided with a relatively small radix, since the number of threads that a thread can communicate with grows logarithmically with system size (e.g., to avoid overflows completely, a radix-8 tree of 512 threads needs at most 31 TTB entries). Receive buffer overflow is avoided by limiting the number of tasks that a victim can send in a burst. Setting this limit to half of the receive buffer size makes overflows rare.

7.4.2 Loop-Parallel runtimes

We adapt the task-parallel runtimes to improve performance with loop-parallel applications. The API is slightly different: Instead of enqueuing and dequeuing tasks, the application enqueues a whole loop and dequeues its iterations. The Carbon runtime uses special support for loop tasks [101]: a loop is enqueued to the GTU in a single enqueue, and loop tasks are partitioned in the GTU. In the software and ADM runtimes, a single task represents a range of loop iterations. Thus, a task can be efficiently split when stealing. Additionally, the runtimes support loops with reductions. The software and Carbon runtimes implement tree-based reductions through shared memory, and in ADM reductions are piggybacked on top of the unblock messages

used to signal phase termination, incurring practically zero overhead.

7.4.3 Discussion

Even though our exploration was not exhaustive and better ADM-based managers may be possible, we draw some important insights about software scheduling for fine-grain parallelism. First, for large-scale CMPs, distributed runtimes are necessary even when ADM is available. Second, the availability of ADM allows all task queues to be kept thread-local. Each queue is accessed by only one thread, either to retrieve its own work or to serve steals. Hence, there is no need for locks or a few expensive misses when stealing occurs between remote threads. For reference, a single remote L2 miss takes around 90 cycles on average in the large-scale CMPs we explored. Third, the overhead of exchanging tasks or other scheduling information through ADM is significantly lower. The latency through the interconnect (25 cycles on average) is typically hidden as the messages are asynchronous. The message handlers in our runtimes typically run in about 50 cycles (including interrupt overhead). Finally, asynchronous messages allow us to overlap scheduling with useful computation.

Since ADM is a general messaging primitive, it could be used to implement other synchronization or communication mechanisms, such as barriers, locks or fast IPC. We leave these further uses to future work.

7.5 Experimental Methodology

Infrastructure: We perform execution-driven simulation of large-scale CMPs using the M5 simulator [17] coupled with the Wisconsin GEMS toolset for memory hierarchy modeling [112]. We simulate user-level application and library code, using detailed microarchitectural models for both the memory hierarchy and the interconnect. All the simulations are performed with warmed-up caches, and we introduce a small random perturbation in the main memory latency and do multiple runs per workload to obtain stable averages [8].

Cores	32–128 cores, 1/2/4 threads per core, in-order, 2-way issue SMT, SPARCv9 ISA, 2 GHz
Coherence	Directory-based, MOESI among L1s-L2s and L2-directories, sequential consistency
L1 caches	32 KB, 4-way set associative, split D/I, 1-cycle latency, private per-core
L2 caches	1 MB per bank, 4 banks/tile, 16-way set associative, non-inclusive, 5-cycle tag / 10-cycle data latencies, pipelined, shared by the L1s of the 4 cores in a tile, crossbar interconnect to L1s
L3 cache	3D-stacked, 16 MB per bank, 1 bank/tile, 16-way set associative, shared across the whole CMP, acts as victim cache for L2s, 10-cycle tag / 21-cycle data latencies, pipelined
Directory	1 bank/tile, idealized 6-cycle latency
MCU	1 memory controller/tile, single DDR-3 channel
Interconnect	2D flattened butterfly, connects tiles of 4 cores
Routers	3-stage pipeline (look-ahead RC and VA, SA, ST [47]), 4 VCs/virtual network, buffering of 8 flits/VC, 3 virtual networks (1 for coherence requests, 1 for ADM/Carbon requests, 1 for replies)
Links	18B flits, repeated and pipelined; 1 cycle latency in local interconnect, 2–11 cycles in global interconnect
Carbon	4-task LTUs, 32 tasks per GTU queue, pipelined GTU
ADM	64-word receive and send buffers (16 4-word messages), 32-entry TTBs

Table 7.3: Main characteristics of the simulated CMPs. The latencies assume a 32 nm process at 2GHz.

Systems: We model tiled CMPs with directory-based cache coherence, focusing on large-scale designs with 32 to 128 cores and a 3-level cache hierarchy, with the parameters shown in Table 7.3. The cores are 2-way in-order similar to the Niagara-2 pipeline [65]. They are also multithreaded to reduce the effect of memory latency. All components are sized to fit under reasonable area and power budgets at 32 nm for the 64-core configuration ($360mm^2$ and 55 W). Area, latency and power of caches and interconnect are estimated using CACTI 5.3 [152], ITRS 2007 predictions, and the models in [12]. The L2s are sized to take 40% of the chip area. We include a 3D-stacked L3, implemented in a 32 nm DRAM process.

Our Carbon model follows the original ISA and microarchitecture [101]. The local task units (LTUs) buffer up to 4 tasks per thread, and can have one task prefetched from the global task unit (GTU). The GTU is located in the center of the CMP, can serve one request per cycle, and holds 32 4-word tasks per thread queue (2KB per thread). The per-core ADM unit uses 64-word send and receive buffers per thread

	Input set	Type	Instrs	Task length	Phase length	Pha-ses	Stolen tasks	L1D hits	L1 misses served by Loc L2	Rem L2	L3	Mem
canneal	native	Loop	2.1 B	4.9 K	657 K	100	2.4 %	94.3 %	14.8 %	34.4 %	36.5 %	14.3 %
mergesort	1M keys	Task	346 M	3.9 K	4.6 M	1	9.8 %	97.3 %	18.5 %	66.1 %	0.0 %	15.4 %
maxflow	4K RLG	Task	1.5 B	674	99 M	1	8.5 %	90.0 %	22.3 %	77.5 %	0.0 %	0.2 %
ced	nyc	Task	381 M	419	3.7 M	2	13.1 %	96.0 %	37.6 %	23.4 %	6.5 %	32.6 %
cg	bcsstk16	Loop	465 M	976	21.8 K	601	7.5 %	90.9 %	77.9 %	20.0 %	0.0 %	2.1 %
gtfold	x54252	Loop	4.3 B	14.8 K	143 K	693	22.2 %	98.3 %	87.1 %	12.3 %	0.0 %	0.6 %
hashjoin	100td3	Task	166 M	1.6 K	3.8 M	1	75.3 %	95.8 %	24.1 %	49.8 %	0.0 %	26.1 %

Table 7.4: Main workload characteristics, using a 64-core CMP with Carbon. The type column indicates whether the application is task or loop-parallel. The average task and parallel phase lengths are in cycles. In loop-parallel applications, task length means iteration length.

(same size as a Carbon queue) and a 32-entry TTB. All other queues for ADM-based runtimes are in software.

Workloads: Our evaluation has three main goals. For balanced applications or codes with sufficiently large tasks, we want to show that the ADM runtime does not introduce any overheads and performs as well as an optimized software-only scheduler. For irregular applications with small tasks that match the scheduling algorithm of Carbon, we want to show that ADM performs and scales as well as Carbon despite using less hardware (queue management and algorithm control in software). Finally, for applications that perform best with other scheduling algorithms, we want to show that the software-mostly nature of ADM runtimes allows us to match the application characteristics and significantly outperform Carbon.

We have selected seven parallel workloads, summarized in Table 7.4. They cover a wide set of domains, use programming models, and exhibit a varied behavior in terms of miss rates, task granularities, available parallelism, and imbalance. They are:

- **canneal:** A loop-parallel circuit routing algorithm using simulated annealing, refactored from the PARSEC suite [16].
- **mergesort:** A parallel implementation of the mergesort algorithm using spawn-sync Cilk-style parallelism. It applies a divide-and-conquer strategy, resorting to serial mergesort when the subarray fits in the L1 cache. Merging two subarrays is parallelized as well.

- **maxflow:** Computes the maximum flow of a graph using the push-relabel algorithm. This graph problem is computationally intensive and has many applications in networking, computer vision, etc., but is hard to scale [11, 100]. It has very short tasks and cores often exhaust their task queues, resulting in frequent stealing.
- **ced:** Performs canny edge detection, a widely used algorithm in image processing and computer vision [28]. Refactored from the OpenCV library.
- **cg:** An iterative solver for sparse linear systems using the conjugate gradient method. Includes different types of phases: sparse matrix-vector multiplications (long but irregular), scaled vector additions (short and regular), and dot products with frequent reductions.
- **gtfold:** A bioinformatics application that predicts the secondary structure of large RNA molecules [113]. It has dependencies between loop iterations and an irregular iteration length, which results in short, imbalanced phases. Tuning the scheduler to the characteristics of this application can yield large benefits.
- **hashjoin:** A hash-join algorithm implementation, common in database workloads [39].

7.6 Evaluation

7.6.1 Software, Carbon and ADM Schedulers

Figure 7.5 shows the performance of software, Carbon and ADM schedulers. Each graph shows the speedup of a single application on CMPs with 32, 64, and 128 dual-threaded cores (64–256 threads). Speedups are normalized to the single-thread software version. This experiment uses the regular software and ADM schedulers explained in Section 7.4 (we only alter the scheduling algorithms in Section 7.6.4). There are several things to observe. First, all applications except maxflow can scale reasonably well. Second, five of the seven benchmarks show large performance differences across the schedulers. Third, there is no single best scheduler across all applications, but ADM performs best on average, being slower than Carbon only on

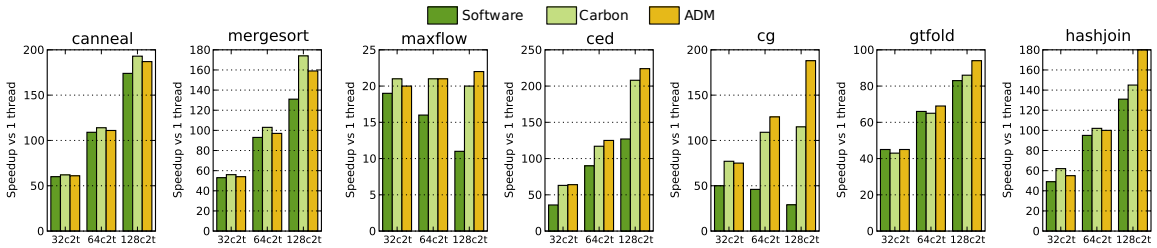


Figure 7.5: Performance of the different fine-grain schedulers, software, Carbon, and ADM, using CMPs with 32, 64 and 128 dual-threaded cores (64–256 threads). Speedups are normalized to the single-thread software version. Higher numbers are better.

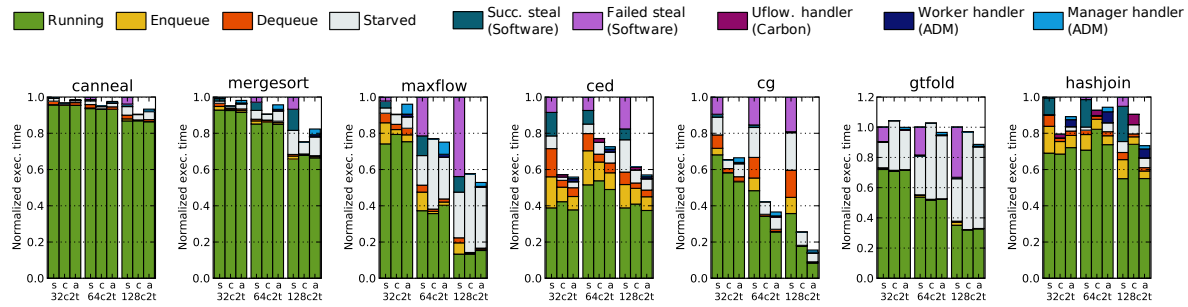


Figure 7.6: Execution time breakdown for software (s) Carbon (c) and ADM (a), using CMPs with 32, 64 and 128 dual-threaded cores (64–256 threads). Each result is normalized to the execution time of the software version *with the same number of cores*. Lower numbers are better.

mergesort.

To gain further insight into these results, Figure 7.6 breaks down execution time into different components on 32–128 dual-threaded cores. Execution time is normalized to the total running time of the software-scheduled version *with the same number of cores*. There are four components common to all runtimes: running (executing non-scheduler application code), enqueuing and dequeuing (from the thread's own queue), and starved (waiting for tasks to become available). For the software scheduler, we also provide the time spent in steals, classified into successful steals (i.e., yielding one or more tasks) and failed steals. For Carbon, we show the amount of time spent

in the underflow handler. For ADM, we show the time spent in interrupt handlers, broken down in the worker and manager portions. We can see some general trends: for Carbon and ADM, most of the scheduling overhead comes from starvation. The ADM overheads from handler code and local enqueues/dequeues are comparatively small, and ADM often beats Carbon by reducing starvation time. For the software scheduler, however, local enqueue/dequeue and stealing overheads can be major, due to loss of locality in the task queues (the software scheduler is non-blocking, so there is no lock contention). We now explain the behavior of each application in detail:

canneal is fairly balanced and coarse-grained, and shows minimal differences between the runtimes.

mergesort, due to its tree-style parallelization, has regions with scarce parallelism, with only a few threads generating new tasks that need to be redistributed as quickly as possible. Thus, mergesort is latency-sensitive, i.e., it significantly benefits from reducing the time that the scheduler takes to distribute work. ADM's fast directed stealing is able to perform within 5% of Carbon in these critical portions, while the software runtime is up to 24% slower.

maxflow does not scale beyond 64 threads, but it is a good example of how efficient scheduling can aid the performance of parallelism-constrained codes. Its tasks are very small (a graph node traversal, ∼600 cycles) and steals are common. ADM and Carbon achieve the same performance. The software runtime is 90% slower at 256 threads, due to enqueue/dequeue and stealing overheads, which become large due to cache misses on queue accesses. Maxflow shows that software queues are inexpensive if kept thread-local: even with 600-cycle tasks, at 64 threads the enqueue/dequeue overheads are 5% in ADM and 4% in Carbon, but 20% in software.

ced has very small tasks (400 cycles), long phases with deep queues and a mild imbalance. ADM and Carbon have similar overheads, both for queuing (because Carbon often reverts to software queues) and load-balancing.

cg combines long (65 K cycles at 128 threads), imbalanced phases, followed by short (4 K cycles at 128 threads), balanced phases, and reductions. ADM matches the performance of Carbon in the short and long phases, but provides tree reductions that are an order of magnitude faster (390 vs 4 K cycles for 128 threads). Reductions

become more important as we increase the number of cores (they are the portion of running time over that of ADM). Additionally, the GTU saturates in the short phases with small tasks, as we will see in Section 7.6.2. These issues cause ADM to outperform Carbon by 70% at 256 threads.

gtfold has relatively large loop iterations: 15 K cycles on average, with a bimodal distribution (either 1 K or 40 K cycles), and has many short, imbalanced parallel phases. Carbon performs sensibly worse than ADM because, even with the GTU disallowing prefetches when there are few tasks, an LTU sometimes prefetches a long task while executing another long task, leading to imbalance since tasks cannot be reclaimed from LTUs. The software and ADM schedulers do not have this problem, but ADM scales better than software.

hashjoin has a large load imbalance (about half of the threads enqueue most of the tasks), causing frequent steals from the empty threads. This frequent stealing saturates the GTU at 256 threads (note the increased overheads due to starvation and overflow handler). Hashjoin also has significant inter-task locality, and since the GTU only steals a small number of tasks per steal (half of the hardware queue size), it produces larger fragmentation, degrading locality and increasing the time spent per task by up to 20%.

To conclude, we discuss the effects of having different stealing policies. In theory, since we use nearest-neighbor stealing in Carbon and software, but hierarchical stealing in ADM, there could be differences in the application locality seen with each runtime. However, excluding hashjoin, the stealing policy has a negligible effect on the execution time of our applications. The only other cases in Figure 7.6 with significant differences in running (non-scheduler) time are maxflow and cg. For maxflow, this is due to algorithmic effects as the amount of work depends on the amount of parallelism. Carbon and ADM can keep more threads busy and end up with more running time on average. For cg, it is due to the faster reductions with ADM. In the remaining applications, the maximum difference between non-scheduler times is 4% (ced), where ADM is slightly faster.

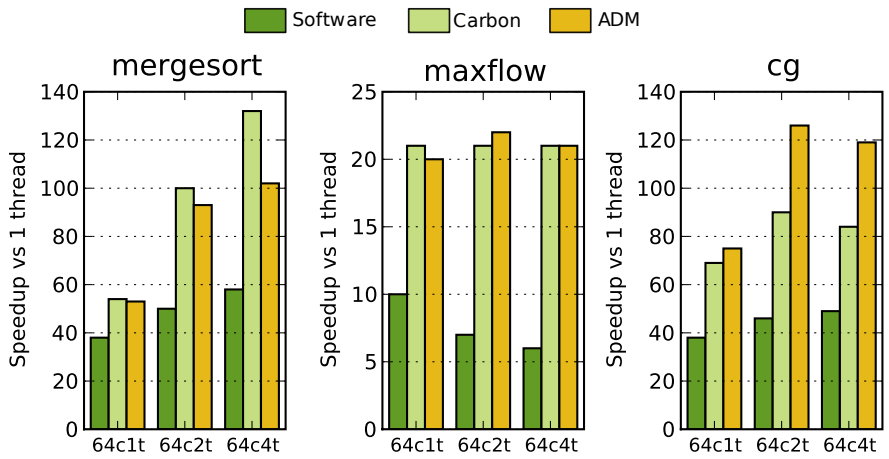


Figure 7.7: Performance of software, Carbon and ADM schedulers on a 64-core CMP with 1, 2 and 4 threads/core.

7.6.2 Sensitivity to Hardware Parameters

Multithreading: Figure 7.7 shows the performance of mergesort, maxflow, and cg using a 64-core CMP with 1, 2 or 4 threads/core (other applications behave similarly). In general, the relative performance differences between software, Carbon, and ADM schedulers remain the same, despite the better latency tolerance with more threads. In mergesort, we note a slight performance reduction for ADM with 4 threads per core. This happens because mergesort is particularly latency-sensitive. With 4 threads, interrupt handlers take more time to execute due to contention in the core's pipeline. This could be addressed by scheduling fewer threads on cores with ADM managers, or prioritizing interrupt handler execution.

Idealized interconnect: Figure 7.8 shows the performance of Carbon and ADM when their traffic is not routed through the conventional interconnect, but through an idealized interconnect with a fixed 25-cycle latency between any source-destination pair and infinite bandwidth. Coherence traffic still uses the conventional interconnect. Additionally, the ideal Carbon GTU serves any number of requests in a single cycle. This allows us to evaluate the effect of *contention* in the GTU. Differences between

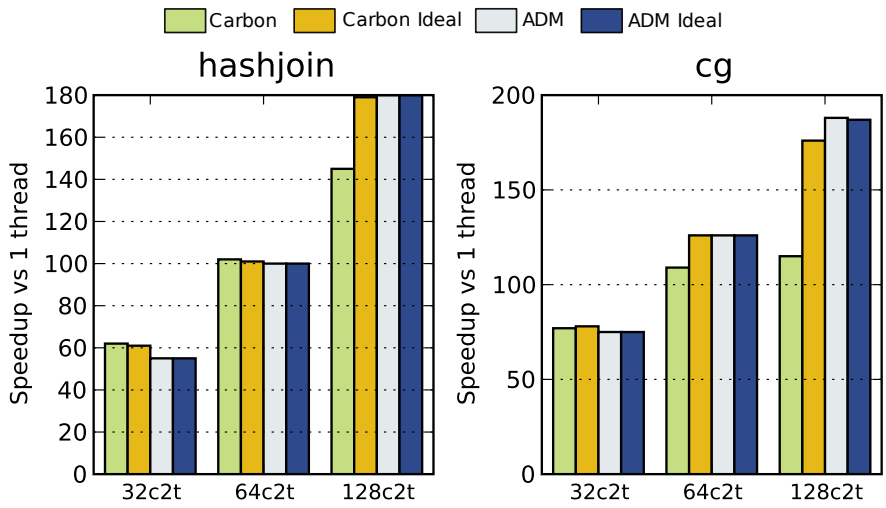


Figure 7.8: Performance of software, Carbon and ADM schedulers when the Carbon/ADM traffic is routed through an idealized network, eliminating contention in the GTU.

ideal and non-ideal configurations are marginal on all applications except hashjoin and cg, where Carbon improves its speed by up to 25% and 55%, respectively, at 256 threads. hashjoin requires very frequent stealing, and the GTU cannot keep up distributing tasks at the required rate. cg has short phases, and the termination messages that the GTU sends at the end of each phase become the bottleneck. Due to its distributed nature, the ADM scheduler is insensitive to these issues. These results indicate that, even with fast hardware, having centralized queues leads to contention with small tasks.

7.6.3 Analysis of ADM Benefits

Since the software and ADM schedulers are fundamentally different, it is hard to understand whether the benefits of ADM come from being able to implement a better scheduling algorithm or from bypassing the memory hierarchy. However, the software runtime cannot be structured in an asynchronous worker/manager organization

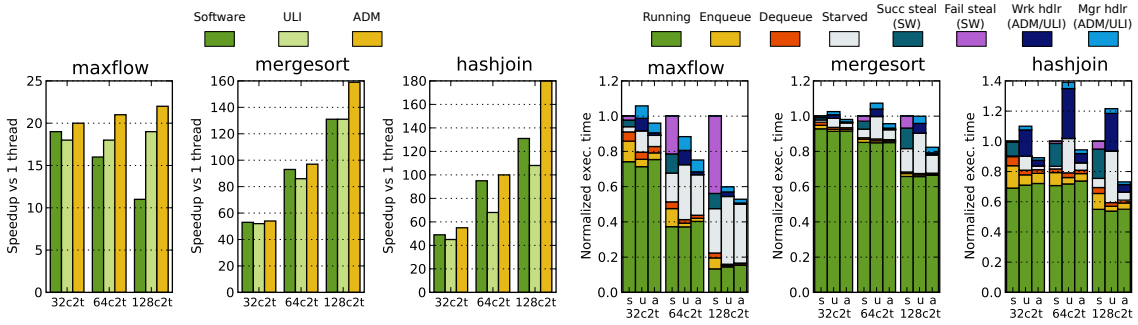


Figure 7.9: Speedup and execution time breakdowns of software (s), ULI (u) and ADM (a) runtimes, with 32–128 cores (64–256 threads).

Operation	ULI			ADM		
	Best	Avg	Worst	Best	Avg	Worst
Enqueue	41	66	84	34	57	72
Dequeue	39	60	94	32	37	44
Recv. update (mgr)	240	263	278	49	52	55
Steal match (mgr)	589	630	703	262	310	365
Send task (victim)	256	403	549	50	76	96
Recv. task (stealer)	259	300	326	40	47	56

Table 7.5: Cost breakdown for ULI and ADM schedulers using a 64-core, 128-thread CMP. The cost for each scheduler operation is given in cycles. Each cost is the average of a specific workload, and includes interrupt overheads for operations triggered by a ULI/message reception. The table includes the best and worst application's costs, and the average cost across all workloads.

without some hardware support. The minimal hardware required is to have *user-level interrupts* (ULI), supported in several recent proposals by monitoring updates to specific cache lines [23, 117, 144]. ULI allows an asynchronous worker/manager scheduler, where one thread can cheaply interrupt another to indicate the availability of a message that includes a task or information on the load of a worker. However, the actual message payload goes through the cache hierarchy. ADM provides *both* asynchronous user-level interrupts and register-to-register messaging.

To understand the benefits of ADM, we implement the same ADM task-parallel scheduler using ULI: threads communicate scheduling events through previously-known cache lines, and notify each other that a message is available using an ULI.

Figure 7.9 shows the speedups and execution time breakdowns of the software, ULI and ADM schedulers for maxflow, mergesort, and hashjoin. ADM always outperforms ULI, since communicating through registers entails lower overheads than through the cache hierarchy. Table 7.5 provides further detail with a cost breakdown of ULI and ADM schedulers. We observe that in ADM schedulers all common operations are fast, taking 37 to 72 cycles on average (matching two threads for a steal takes longer, but occurs relatively rarely). With ULI, the costs for enqueuing and dequeuing are only slightly higher than in ADM, since task queues are thread-local and update messages are sent only at specific thresholds. However, costs for the other operations, which always involve sending or receiving messages, are two to six times larger than with ADM. Sending or receiving a message with ULIs requires at least two cache-to-cache transfers (one for the data and one for the ULI), taking up to 200 cycles (the penalty is lower if sender and receiver share the L2). Moreover, if the application is sensitive to scheduler latency, these higher overheads will increase starvation. The results in Figure 7.9 illustrate how these issues affect each application. In maxflow, which is somewhat latency-sensitive, ULI is only 18% slower than ADM. In mergesort, ADM achieves higher scalability than ULI because it can perform steals faster, reducing starvation. Finally, in hashjoin the ADM/ULI schedulers typically steal several tasks at once, causing multiple messages to be sent per handler execution. As a result, the handler overhead increases by up to 4× in ULI, being even slower than the software scheduler.

In conclusion, while ULI can be a useful mechanism for schedulers, ADM outperforms ULI by large margins. Moreover, the benefits of ADM versus a normal, synchronous software scheduler come from *both* being able to implement an asynchronous scheduler and bypassing the cache hierarchy, with the relative importance of each cause being application-dependent.

7.6.4 Using Custom Scheduling Algorithms

We now use gtfold to illustrate the potential of adapting the scheduling algorithm to the application. This application operates over an upper triangular matrix, and

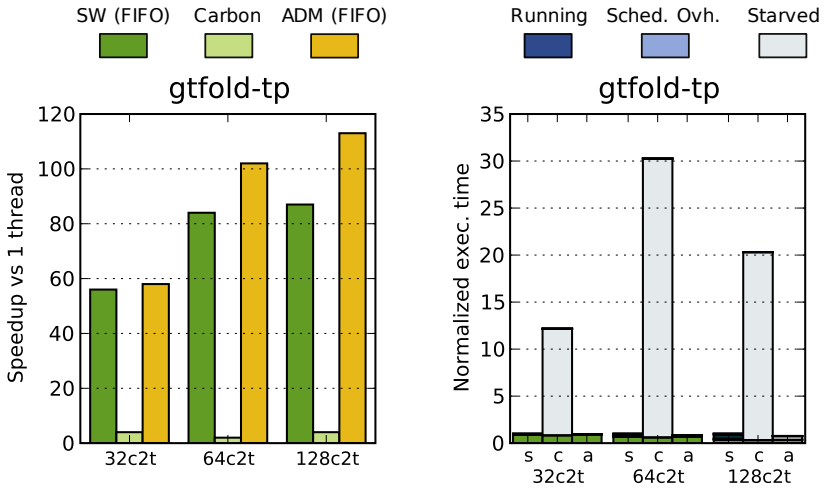


Figure 7.10: Speedup and execution time breakdowns of software (s), Carbon (c) and ADM (a) runtimes for the task-parallel version of gtfold, with 32–128 dual-thread CPUs (64–256 threads). Software and ADM schedulers are modified to use FIFO queues.

has non-trivial dependencies: task (i, j) depends on $(i, j-1)$ and $(i+1, j)$. The original application is loop-parallel, scheduling one diagonal per phase to avoid these dependencies. Since tasks also have a bimodal distribution (being either 1 K or 38 K cycles), this leads to short, imbalanced phases. However, we can avoid having multiple parallel phases by refactoring gtfold as a task-parallel application: each task works on a single element of the matrix, and every time a task completes, it checks if it is the last dependent task to complete for each of its successors, and enqueues them if so. This can be done at low overhead with counters and atomic operations through shared memory, since tasks are 15 K cycles on average. Ideally, we want to execute the tasks that have a longer dependence chain first, which roughly corresponds to a FIFO enqueuing policy. LIFO will do poorly, since it keeps older tasks (with higher potential to clear critical dependencies) at the bottom of the queue.

Figure 7.10 shows the speedups and execution time breakdowns for the refactored gtfold. The software and ADM schedulers are modified to perform FIFO enqueues and dequeues, while Carbon retains its LIFO policy. Software and ADM achieve

significant performance improvements over the loop-parallel versions in Figure 7.5 (35% for ADM at 256 threads). ADM still outperforms software by 40%. In contrast, Carbon achieves a maximum speedup of only 3× over the sequential version, being 40× slower than ADM. With 256 threads, the task-parallel version of gtfold on ADM outperforms the loop-parallel version on Carbon by 50%.

This example demonstrates the benefits of being able to implement scheduling algorithms in software. While Carbon could also implement FIFO queues in hardware, this would increase its design and verification complexity. Given the large number of scheduling algorithms that can be useful, supporting them all in hardware is infeasible. For instance, hashjoin benefits from directed hierarchical stealing, cg requires fast reductions, and other programming models need more complex queuing policies (e.g., our GRAMPS runtime from Chapter 6 requires one task queue per stage for the scheduler, and one packet queue per size bin for the buffer manager). Our results show that supporting a simple, yet flexible primitive like ADM and leaving algorithmic decisions to software is more practical and leads to better performance than implementing them in hardware.

7.7 Additional Related Work

ADM is inspired by previous efforts in architectures that integrate shared memory and message passing. UDM [111], the messaging system in Alewife/FUGU [1], implements a model similar to ADM in some aspects. UDM supports low-overhead short messages, which can be received synchronously or asynchronously via user-level interrupts (with around a 100-cycle interrupt overhead). User-level code can disable these user-level interrupts, and threads transparently buffer received messages in memory with a privileged interrupt that triggers when a handler or atomic section takes too long. Like UDM, ADM includes asynchronous message receive through user-level interrupts, but at a lower overhead since no privileged interrupt handling code needs to run. ADM uses a similar mechanism to UDM to back the limited receive-side buffers with unbounded queues in software, but it is based on receiver-side buffer occupancy,

not on timeouts. Finally, in ADM messages are sent to virtual threads, not physical thread contexts, allowing the OS to perform thread migration and more flexible scheduling. StarT-Voyager [9] implements user-level message passing by exposing memory-mapped send and receive queues that can overflow to main memory, but these memory-mapped queues entail additional overheads. The J-Machine [120] and M-Machine [106] also include a set of flexible messaging mechanisms suitable for fine-grain asynchronous communication, but unlike these message-driven architectures, for which messaging is the main means of communication, we advocate introducing messaging support in a shared-memory CMP.

Several recent architecture proposals target scheduling issues. Apart from Carbon, researchers have proposed several hardware schedulers that are tailored to specific applications or hardware [6, 56, 142]. Rigel [90] is a large-scale accelerator CMP design with incoherent shared memory that includes a globally shared cache and special support for atomic operations to improve the efficiency of task-parallel software runtimes. However, Rigel targets task sizes one to two orders of magnitude larger than we do (100 K cycles per task). Pangaea [23, 160] is a tightly integrated small scale CPU-GPU design in which a CPU core dispatches work to GPU cores using user-level interrupts (ULIs). As we have shown, ULI-based schedulers can suffer large performance penalties because communication still happens through shared memory.

In the context of shared memory multiprocessors and CMPs, there have been several proposals to accelerate synchronization primitives using message-like constructs. Decoupled software pipelining [131] uses synchronous producer-consumer queues between processors for fine-grain parallelization of sequential programs. HAQu [105] implements hardware-accelerated single-producer single-consumer queues. QOLB [87] focuses on reducing locking overhead, with hardware that maintains a distributed queue and performs direct node-to-node lock transfers. Active Memory Operations [57] use messages between cores and memory controllers, which are augmented with some extra logic, to implement fast locks and barriers. We note that ADM could also be used to implement these primitives, and leave the detailed evaluation to future work.

7.8 Summary

This chapter has presented a hardware-software approach to build efficient fine-grain schedulers on large-scale CMPs. We propose asynchronous direct messages (ADM), a flexible and practical messaging mechanism that allows threads to communicate scheduling information without going through the memory hierarchy. Using ADM, we develop scalable schedulers that keep all scheduling data structures, such as task queues, thread-local, even when stealing occurs, and overlap most communication with useful computation. We show that ADM-based schedulers clearly outperform software-only schedulers and match or exceed the performance of hardware-only Carbon. When the ADM scheduler tailors its scheduling algorithm to the application characteristics, it exceeds Carbon's performance by up to 70%. Our results show that supporting a simple, yet flexible primitive like ADM and leaving the algorithmic decisions to software is more practical and leads to better performance than implementing scheduling in hardware.

Chapter 8

Conclusions and Future Work

This dissertation has presented hardware and software techniques to enable efficient, scalable and easy to use CMPs with hundreds to thousands of cores. In particular, we have made the following contributions:

- **Scalable Memory Hierarchies:** We have designed three techniques that, together, enable cache hierarchies to scale to thousands of cores efficiently. First, ZCache (Chapter 3) provides high associativity at low cost and is characterized with simple and accurate workload-independent models. We use the high associativity and analytical models of ZCache to develop two techniques that solve crucial scalability problems on the shared components of the memory hierarchy. Vantage (Chapter 4) implements scalable and efficient fine-grain cache partitioning, which enables hundreds of threads to share caches in a controlled fashion, providing configurability, isolation and QoS guarantees. SCD (Chapter 5) is a coherence directory that scales to thousands of cores efficiently and causes negligible directory-induced invalidations with minimal overprovisioning, enabling efficient cache coherence with QoS guarantees in large-scale CMPs.
- **Scalable Dynamic Fine-Grain Runtimes:** We have developed a set of techniques that enable efficient dynamic runtimes and schedulers for programming models with rich semantics that scale well. First, we have developed techniques to perform dynamic fine-grain scheduling of streaming workloads, and evaluated

them by building the first runtime for the GRAMPS programming model (Chapter 6). This runtime leverages programming model information about parallelism, task dependencies and priorities, and producer-consumer communication to improve scheduling, guarantees memory footprint, and outperforms previous schedulers (both static and dynamic) on a wide range of applications. Second, we have developed a hybrid hardware-software scheme to accelerate dynamic schedulers and make them scale efficiently into the hundreds of cores (Chapter 7), even under frequent communication and synchronization. We have designed ADM, a messaging primitive tailored to the needs of scheduling and control applications, and used it to build scalable and efficient hardware-accelerated schedulers that match or outperform hardware-only schedulers while retaining the flexibility of software schedulers.

We believe that these contributions open several interesting avenues for future research. Our work in scaling the memory hierarchy places an important emphasis on using analytical models throughout the design process, and shows that, contrary to conventional wisdom, this design style has several benefits over a purely empirical approach: our techniques not only perform better in the common case, but can make strict guarantees in all scenarios, which is crucial to provide performance isolation and QoS guarantees. We hope that architects apply this design approach to other system components, and especially shared resources, so that we can achieve computing systems that provide end-to-end QoS guarantees, enabling full isolation among concurrent applications, predictable performance, and efficient and coordinated use of shared resources. Additionally, several of our contributions provide new capabilities to software, and enable interesting new schemes that are beyond the scope of this dissertation. For example, a scheduler could use Vantage's scalable cache partitioning to strictly control its on-chip memory footprint, and use feedback from Vantage to adapt its footprint (e.g., by changing queue sizes and partition sizes) to optimize its performance. We leave all these endeavors to future work.

Bibliography

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995. 133, 159

[2] A. Agarwal and S. D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993. 12, 25

[3] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of the 15th annual Intl. Symp. on Computer Architecture*, 1988. 17, 82

[4] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *Proc. of the 19th annual ACM Symp. on Parallel Algorithms and Architectures*, 2007. 20, 117

[5] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000. 1, 8

[6] G. Al-Kadi and A. S. Terechko. A hardware task scheduler for embedded video processing. In *Proc. of the 4th intl. conf. on High Performance and Embedded Architectures and Compilers*, 2009. 160

[7] A. Alameldeen and D. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006. 40

[8] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. of the 9th intl. symp. on High-Performance Computer Architecture*, 2003. 147

[9] B. Ang, D. Chiou, L. Rudolf, and Arvind. Message passing support on StarT-Voyager. In *Proc. of the 5th intl. conf. on High Performance Computing*, 1998. 160

[10] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of the 10th annual ACM Symp. on Parallel Algorithms and Architectures*, 1998. 109, 117, 134

[11] D. A. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proc. 18th intl. conf. on Parallel and Distributed Computing Systems*, 2005. 150

[12] J. Balfour and W. J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *Proc. of the 20th annual Intl. Conf. on Supercomputing*, 2006. 148

[13] L. 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 *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000. 17, 82, 89

[14] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *Proc. of the 40th annual IEEE/ACM intl symp. on Microarchitecture*, 2007. 12, 13, 26

[15] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2), 1966. 34

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. of the 17th intl. conf. on Parallel Architectures and Compilation Techniques*, 2008. 9, 40, 96, 149

[17] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006. 147

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970. 32

[19] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. of the 35th annual symp. on Foundations of Computer Science*, 1994. 20, 110, 117

[20] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995. 25, 38

[21] M. Bohr. Interconnect scaling: The real limiter to high-performance ULSI. In *Proc. of the Intl. Electron Devices Meeting*, 1995. 8

[22] S. Borkar. Thousand core chips: a technology perspective. In *Proc. of the 44th annual Design Automation Conf.*, 2007. 1, 9

[23] A. Bracy, K. Doshi, and Q. Jacobson. Disintermediated active communication. *Comput. Archit. Lett.*, 5(2), 2006. 156, 160

[24] M. Brehob, S. Wagner, E. Torng, and R. Enbody. Optimal replacement is NP-Hard for nonstandard caches. *IEEE Trans. Comput.*, 53(1), 2004. 43

[25] M. W. Brehob. *On the Mathematics of Caching*. PhD thesis, Michigan State University, 2003. 26

[26] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In

Proc. of the 20th intl. conf. on Parallel Architectures and Compilation Techniques, 2011. 19, 21, 107

[27] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proc. of the 2nd IEEE symp. on High-Performance Computer Architecture*, 1996. 12, 13, 26

[28] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6), 1986. 150

[29] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proc. of the 32nd annual Intl. Symp. on Computer Architecture*, 2005. 83

[30] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical report, Center for Computing Sciences, IDA, 1999. 130

[31] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proc. of the 9th annual ACM Symp. on Theory of Computing*, 1977. 30, 70, 94

[32] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd annual Intl. Symp. on Computer Architecture*, 2006. 14, 61

[33] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proc. of the 4th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 1991. 17, 82

[34] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th annual ACM SIGPLAN conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2005. 20, 109

[35] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proc. of the 17th annual ACM Symp. on Parallel Algorithms and Architectures*, 2005. 117, 120, 134, 142

[36] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009. 47

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2009. 123

[38] J. Chen, M. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand. A reconfigurable architecture for load-balanced rendering. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics Hardware*, 2005. 130

[39] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proc. of the 19th annual ACM Symp. on Parallel Algorithms and Architectures*, 2007. 150

[40] X. Chen, Y. Yang, G. Gopalakrishnan, and C. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer Aided Design*, 2006. 83

[41] X. Chen, Y. Yang, G. Gopalakrishnan, and C. Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Formal Methods in System Design*, 36(1), 2010. 83

[42] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proc. of the 37th annual Design Automation Conf.*, 2000. 15, 49, 50, 51, 60, 61

[43] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proc. of the 36th annual IEEE/ACM intl. symp. on Microarchitecture*, 2003. 47

[44] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proc. of the 37th Intl. Conf. on Parallel Processing*, 2008. 142

[45] H. Cook, K. Asanović, and D. A. Patterson. Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. Technical report, EECS Department, U. of California, Berkeley, 2009. 14, 60, 61

[46] W. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proc. of the 38th annual Design Automation Conf.*, 2001. 138

[47] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003. 138, 148

[48] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Proc. of the 17th annual Intl. Conf. on Supercomputing*, 2003. 111

[49] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah. A 22nm IA multi-CPU and GPU System-on-Chip. In *IEEE Intl. Solid-State Circuits Conf.*, 2012. 1, 9

[50] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proc. of the 15th intl. conf. on Parallel Architectures and Compilation Techniques*, 2006. 20, 121

[51] S. Demetriades, M. Hanna, S. Cho, and R. Melhem. An efficient hardware-based multi-hash scheme for high speed IP lookup. In *Proc. of the 16th IEEE symp. on High Performance Interconnects*, 2008. 46

[52] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), 1974. 7

[53] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *4th Intl. Workshop in OpenMP*, 2008. 20, 109, 110

[54] N. Enright Jerger, L. Peh, and M. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *Proc. of the 41st annual IEEE/ACM intl. symp. on Microarchitecture*, 2008. 83

[55] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark Silicon and The End of Multicore Scaling. In *Proc. of the 38th annual Intl. Symp. on Computer architecture*, 2011. 8, 9

[56] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. Badia, E. Ayguade, J. Labarta, and M. Valero. Task Superscalar: An Out-of-Order Task Pipeline. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2010. 160

[57] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *Proc. of the 21st annual Intl. Conf. on Supercomputing*, 2007. 160

[58] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proc. of the 2006 ACM/IEEE conf. on Supercomputing*, 2006. 130

[59] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A scalable directory for many-core systems. In *Proc. of the 17th IEEE intl. symp. on High Performance Computer Architecture*, 2011. 17, 80, 82, 83, 88, 93, 96, 103

[60] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 1998. 19, 106, 109, 120, 121, 134

[61] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44(1), 2011. 8

[62] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A sub-1W to 2W low-power IA processor for mobile internet devices and ultra-mobile PCs in 45nm hi-K metal gate CMOS. In *IEEE Intl. Solid-State Circuits Conf.*, 2008. 39, 68, 94

[63] S. Ghemawat and P. Menage. TCMalloc: Thread-Caching Malloc http://goog-perftools.sourceforge.net/doc/tcmalloc.html. 129

[64] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of the 12th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2006. 111, 113, 121

[65] G. Grohoski. Niagara2: A highly-threaded server-on-a-chip. In *18th Hot Chips Symp.*, 2006. 148

[66] S. Guo, H. Wang, Y. Xue, C. Li, and D. Wang. Hierarchical Cache Directory for CMP. *Journal of Computer Science and Technology*, 25(2), 2010. 82

[67] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for terminally strict parallel programs. In *Proc. of the 23rd IEEE Intl. Parallel and Distributed Processing Symp.*, 2009. 110, 129

[68] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proc. of the 15th ACM SIGPLAN symp. on Principles and Practice of Parallel Programming*, 2010. 129

[69] A. Gupta, W. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proc. of the 19th Intl. Conf. on Parallel Processing*, 1990. 17, 82

[70] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000. 13, 26

[71] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture*, 2004. 14, 61

[72] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4), 2011. 9

[73] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (5th ed.)*. Morgan Kaufmann, 2011. 81

[74] Hewlett-Packard. Inside the Intel Itanium 2 processor. Technical report, 2002. 33

[75] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12), 1989. 34

[76] R. Hoffmann, M. Korch, and T. Rauber. Performance Evaluation of Task Pools Based on Hardware Synchronization. In *Proc. of the 2004 ACM/IEEE conf. on Supercomputing*, 2004. 109, 110

[77] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proc. of the 18th intl. conf. on Parallel Architectures and Compilation Techniques*, 2009. 130

[78] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proc. Symp. on Interactive 3D Graphics and Games*, 2007. 130

[79] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32

message-passing processor with DVFS in 45nm CMOS. In *IEEE Intl. Solid-State Circuits Conf.*, 2010. 1

[80] L. Hsu, S. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. of the 15th intl. conf. on Parallel Architectures and Compilation Techniques*, 2006. 14

[81] Intel. TBB http://www.threadingbuildingblocks.org. 19, 106, 109, 130

[82] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proc. of the 18th annual intl. conf. on Supercomputing*, 2004. 50, 52, 67

[83] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. of the 17th intl. conf. on Parallel Architectures and Compilation Techniques*, 2008. 47, 69, 77

[84] A. Jaleel, M. Mattina, and B. Jacob. Last Level Cache (LLC) Performance of Data Mining Workloads On A CMP. In *Proc. of the 12th intl. symp. on High-Performance Computer Architecture*, 2006. 96

[85] A. Jaleel, K. Theobald, S. C. S. Jr, and J. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proc. of the 37th annual Intl. Symp. on Computer Architecture*, 2010. 33, 40, 47, 64, 77

[86] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the 17th annual Intl. Symp. on Computer Architecture*, 1990. 12, 26

[87] A. Kägi, D. Burger, and J. R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proc. of the 24th annual Intl. Symp. on Computer Architecture*, 1997. 160

[88] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *Proc. of the ACM SIGPLAN conf. on Languages, Compilers, and Tools for Embedded Systems*, 2003. 111, 121

[89] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20, 1998. 121

[90] J. Kelm, D. Johnson, M. Johnson, N. Crago, W. Tuohy, A. Mahesri, S. Lumetta, M. Frank, and S. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009. 13, 15, 94, 160

[91] J. Kelm, M. Johnson, S. Lumetta, and S. Patel. WayPoint: scaling coherence to 1000-core architectures. In *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010. 16, 17, 83, 94

[92] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: media processing with streams. *Micro, IEEE*, 21(2), 2001. 13, 111

[93] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proc. of the 10th intl. symp. on High-Performance Computer Architecture*, 2004. 13, 25

[94] Khronos Group. OpenCL 1.0 specification, 2009. 19, 106, 110

[95] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2002. 47

[96] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. In *Proc. of the European Symp. on Algorithms*, 2008. 88

[97] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. of the 8th annual Intl. Symp. on Computer Architecture*, 1981. 31

[98] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 2008. 111, 118, 121

[99] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, 2008. 109, 110

[100] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proc. of the 20th annual Symp. on Parallelism in Algorithms and Architectures*, 2008. 150

[101] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proc. of the 34th annual Intl. Symp. on Computer Architecture*, 2007. 21, 132, 134, 142, 146, 148

[102] G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010. 1, 9, 17, 82, 94, 95

[103] A. Lebeck and D. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proc. of the 22nd annual Intl. Symp. in Computer Architecture*, 1995. 99

[104] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1), 1987. 20, 113

[105] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *Proc. of the 17th intl. symp. on High-Performance Computer Architecture*, 2011. 160

[106] W. S. Lee, W. Dally, S. Keckler, N. Carter, and A. Chang. An efficient, protected message interface. *IEEE Computer*, 31(11), Nov 1998. 160

[107] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *Proc. of the 34th annual Intl. Symp. on Computer Architecture*, 2007. 13

[108] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009. 39, 95

[109] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. of the 14th IEEE intl. symp. on High Performance Computer Architecture*, 2008. 15, 50, 51

[110] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 2005. 39

[111] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. Kaashoek. Exploiting two-case delivery for fast protected messaging. In *Proc. of the 4th intl. symp. on High-Performance Computer Architecture*, 1998. 133, 159

[112] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005. 147

[113] A. Mathuriya, D. A. Bader, C. E. Heitsch, and S. C. Harvey. GTfold: A scalable multicore code for RNA secondary structure prediction. In *Proc. of the 2009 ACM Symp. on Applied Computing*, 2009. 150

[114] M. Mitzenmacher. Some open questions related to cuckoo hashing. In *Proc. of the 17th annual European Symp. on Algorithms*, 2009. 46

[115] G. E. Moore. Cramming more components onto integrated circuits. 38(8), 1965. 7

[116] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture*, 2007. 11, 39

[117] V. Nagarajan and R. Gupta. ECMon: exposing cache events for monitoring. In *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009. 14, 156

[118] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *Proc. of the 18th intl. conf. on Parallel Architectures and Compilation Techniques*, 2009. 130

[119] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *Proc. of the 16th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2011. 110

[120] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine multicomputer: an architectural evaluation. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993. 160

[121] NVIDIA. CUDA SDK Code Samples `http://developer.nvidia.com/object/cuda_sdk_samples.html`. 123

[122] NVIDIA. CUDA 3.0 reference manual, 2010. 19, 106, 110

[123] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS VII*, 1996. 8

[124] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proc. of the 9th annual European Symp. on Algorithms*, 2001. 24, 32, 46

[125] J. Park and W. J. Dally. Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures. In *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, 2010. 111, 118, 121

[126] C. Percival. Cache missing for fun and profit. *BSDCan*, 2005. 14

[127] M. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proc. of the 10th intl. symp. on High-Performance Computer Architecture*, 2009. 14

[128] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006. 14, 50, 67, 68, 69

[129] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proc. of the 32nd annual Intl. Symp. on Computer Architecture*, 2005. 13, 26

[130] J. Ragan-Kelley. Keeping Many Cores Busy: Scheduling the Graphics Pipeline. In *SIGGRAPH Courses*, 2010. 21

[131] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proc. of the 13th intl. conf. on Parallel Architectures and Compilation Techniques*, 2004. 160

[132] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000. 15, 50, 51

[133] D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive line placement with the set balancing cache. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009. 12, 26

[134] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2010. 3

[135] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. of the 38th annual Intl. Symp. in Computer Architecture*, 2011. 4

[136] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proc. of the 18th intl. symp. on High Performance Computer Architecture*, 2012. 1, 4, 9

[137] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *Proc. of the 20th intl conf. on Parallel Architectures and Compilation Techniques*, 2011. 5

[138] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture*, 2007. 30

[139] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *Proc. of the 15th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2010. 6

[140] A. Seznec. A case for two-way skewed-associative caches. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993. 13, 24, 25, 26

[141] 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 *Intl. Solid-State Circuits Conf.*, 2010. 1, 9, 15, 51

[142] M. Själander, A. Terechko, and M. Duranton. A look-ahead task management unit for embedded multi-core architectures. In *Proc. of the 11th EUROMICRO Conf. on Digital System Design Architectures, Methods and Tools*, 2008. 160

[143] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference*. MIT Press, 1998. 130

[144] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-update: a communication aid for shared memory multiprocessors. In *Proc. of the 12th ACM SIGPLAN symp. on Principles and Practice of Parallel Programming*, 2007. 156

[145] J. Sugerman. *Programming Many-Core Systems With GRAMPS*. PhD thesis, Stanford University, 2010. 111

[146] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1), 2009. 21, 107, 112, 122

[147] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. of the 8th IEEE intl. symp. on High Performance Computer Architecture*, 2002. 14

[148] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010. 130

[149] Sun Microsystems. UltraSPARC T2 supplement to the UltraSPARC architecture 2007. Technical report, 2007. 17, 25, 33, 82

[150] Y. Taur and E. Nowak. CMOS devices below 0.1um: How high will performance go? In *Proc. of the Intl. Electron Devices Meeting*, 1997. 8

[151] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the 10th Intl. Conf. on Compiler Construction*, 2001. 19, 21, 106, 107, 111, 121, 123

[152] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008. 148

[153] Tilera. TILE-Gx 3000 Series Overview. Technical report, 2011. 1, 9, 15, 94

[154] S. Tzeng, A. Patney, and J. Owens. Task management for irregular-parallel workloads on the GPU. In *Proc. of the conf. on High Performance Graphics*, 2010. 130

[155] K. Varadarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions. In *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006. 50, 51

[156] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th annual Intl. Symp. on Computer Architecture*, 1992. 139

[157] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring it all to software: Raw machines. *IEEE Computer*, 30(9), 1997. 13

[158] D. A. Wallach. PHD: A Hierarchical Cache Coherent Protocol. Master's thesis, Massachusetts Institute of Technology, 1992. 17, 82

[159] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S. Chu, S. Islam, and V. Zyuban. The implementation of POWER7: A highly parallel and scalable multi-core high-end server processor. In *IEEE Intl. Solid-State Circuits Conf.*, 2010. 1, 9

[160] 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 *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008. 160

[161] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995. 96

[162] C. Wu and M. Martonosi. A Comparison of Capacity Management Schemes for Shared CMP Caches. In *Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, 2008. 15, 50, 52

[163] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009. 15, 47, 50, 52, 67, 68, 69

[164] Q. Yang, G. Thangadurai, and L. Bhuyan. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3), 1992. 17, 82

[165] S.-H. Yang, S. Lee, J. Y. Lee, J. Cho, H.-J. Lee, D. Cho, J. Heo, S. Cho, Y. Shin, S. Yun, E. Kim, U. Cho, E. Pyo, M. H. Park, J. C. Son, C. Kim, J. Youn, Y. Chung, S. Park, and S. H. Hwang. A 32nm high-k metal gate application processor with GHz multi-core CPU. In *Intl. Solid-State Circuits Conf.*, 2012. 1, 9

[166] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *Proc. of the Workshop on Java for High-Performance Network Computing*, 1998. 130

[167] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2009. 123

[168] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A Fully Integrated Multi-CPU, GPU and Memory Controller 32nm Processor. In *Intl. Solid-State Circuits Conf.*, 2011. 10

[169] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture*, 2007. 83

[170] J. Zebchuk, V. Srinivasan, M. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009. 17, 83

[171] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. *IEEE Micro*, 17(5), 1997. 26

[172] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing pattern-based directory coherence for multicore scalability. In *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010. 17, 83