The Dissertation Committee for Jeffrey Robert Diamond
certifies that this is the approved version of the following dissertation:

Designing On-chip Memory Systems for Throughput Architectures

Committee:

Donald S. Fussell, Supervisor

Stephen W. Keckler, Co-Supervisor

Robert van de Geijn

Calvin Lin

Victor Eijkhout

Designing On-chip Memory Systems for Throughput Architectures

by

Jeffrey Robert Diamond, B.S.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2015

For my wife, Iris, and my mother, Irene.

Acknowledgments

I'd like to thank the many people who helped me on this long and winding road that resulted in this dissertation.

First I'd like to thank my co-advisor, Don Fussell. No other single person invested so much time in my education and had so much faith in its worth. Don acted like a father to me, spending thousands of valuable hours teaching me the facts of life, from computer graphics to the great truths of human behavior. Above all else, he refused to let me fail. Next I'd like to thank my other co-advisor, Steve Keckler. I relished the many talks we had about computer architecture, and I thank Steve for the considerable funding he gave my work. Steve is an inspiration by self example, always professional and fair.

Next, I'd like to thank my committee members. Robert van de Geijn was my long time tutor in linear algebra, and his class in matrix multiplication was one of my favorites. Robert always gave me advice when no one else would, and was inspiring in the way he helped others. Calvin Lin was always available to me for discussing architectural ideas, and I'm very grateful for the wonderful place he gave me to work. Calvin is also a master of writing skills, and I am grateful for his writing tips, even if they are not evident in this work. Finally, Victor Eijkhout knew how to say the hard things I needed to hear, without actually holding them against me.

Along the way, I'd like to thank Warren Hunt, whose ACL2 project was a precursor to AMI, J. C. Brown, who funded and pushed through the HOMME work,

and Martin Burtscher, who showed me what it meant to be a professor. I'd also like to thank Lydia Griffith, who helped me through the formal process of getting a PhD.

I am also grateful for all the students who helped me along the way. Behnam Robatmili was my first co-author and closest academic companion. Paul Gratz, Walter Chang, Paul Navratil, and Peter Djeu took me under their wings and taught me the ropes of grad school. Bert Maher and Katie Coons provided unflagging emotional support at both the beginning, middle, and end of my grad school career, and really helped when I needed it most. Sean Keely has been an incredible friend and ally in that special zone of real world hardware implementation, and I have really enjoyed the technical discussions and advice. I'd also like to thank the students who kept me company and gave me morale support in the final end game - Akanksha Jain, Curtis Dunham, Ashay Rane and Sarah Abraham, as well as the other GSEM folks, Joao Barbosa, Randy Smith, Christian Miller, Andrew Dreher, and countless others.

Finally, I'd like to thank my wife, Iris. This was mostly her idea, and she bore most of the cost. I'd also like to thank my mother, Irene, who constantly supported my dissertation, but did not live to see its completion.

Jeffrey Robert Diamond

The University of Texas at Austin

December 2015

Designing On-chip Memory Systems for Throughput Architectures

Publication No. _____

Jeffrey Robert Diamond, Ph.D.
The University of Texas at Austin, 2015

Supervisors: Donald S. Fussell
Stephen W. Keckler

Driven by the high arithmetic intensity and embarrassingly parallel nature of real time computer graphics, GPUs became the first wide spread throughput architecture. With the end of Dennard scaling and the plateau of single thread performance, nearly all computer chips at all scales have now become explicitly parallel, containing a hierarchy of cores and threads. Initially, these individual cores were imagined to be no different from traditional uniprocessors, and parallel programs no different than traditional parallel programs. Like GPUs, these modern chips share finite on-chip resources between threads. This results in novel performance and optimization issues at any granularity of parallelism, from cell phones to GPUs.

Unfortunately, the performance characteristics of these systems tend to be non-linear and counter-intuitive. The programmer's software stack has been slow in adapting to this paradigm shift. Compilers still focus primarily on optimizing single thread performance at the expense of throughput. Existing parallel applications are not a perfect match for modern multicore, multithreaded processors. And existing methodologies for performance analysis and simulation are not aligned with multicore issues.

This dissertation begins with a mathematical analysis of throughput performance in the presence of shared on-chip resources. When cache hit rates begin to fall, there is a steep drop off in throughput performance. An optimistic view of this regime is that even small improvements to cache efficiency offer significant benefits. This motivates the exploration of general throughput optimizations in both hardware and software that apply to both coarse-grained and fine-grained parallel architectures, requiring no programmer intervention or tuning. This dissertation provides two such solutions.

The first solution is a compiler optimization called "loop microfission" that can boost throughput performance by up to 50%. In the context of the intrachip scalability of supercomputing applications, we demonstrate the failings of conventional performance tuning software and compiler algorithms in the presence of shared resources. We introduce a new approach to throughput optimization, including a memory friendly performance analysis tool, and show that techniques for throughput optimization are similar to traditional optimizations, but require new priorities.

The second solution is a hardware optimization called Arbitrary Modulus Indexing (AMI), a technique that generalizes efficient implementations of the DIV/-MOD operation from Mersenne Primes to all integers. We show that the primary performance bottlenecks in modern GPUs for regular, memory intensive applications are bank and set conflicts in the shared on-chip memory system. AMI completely

eliminates conflicts in all facets of the memory system at negligible hardware cost, and has even broader potential for optimizations throughout computer architecture.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

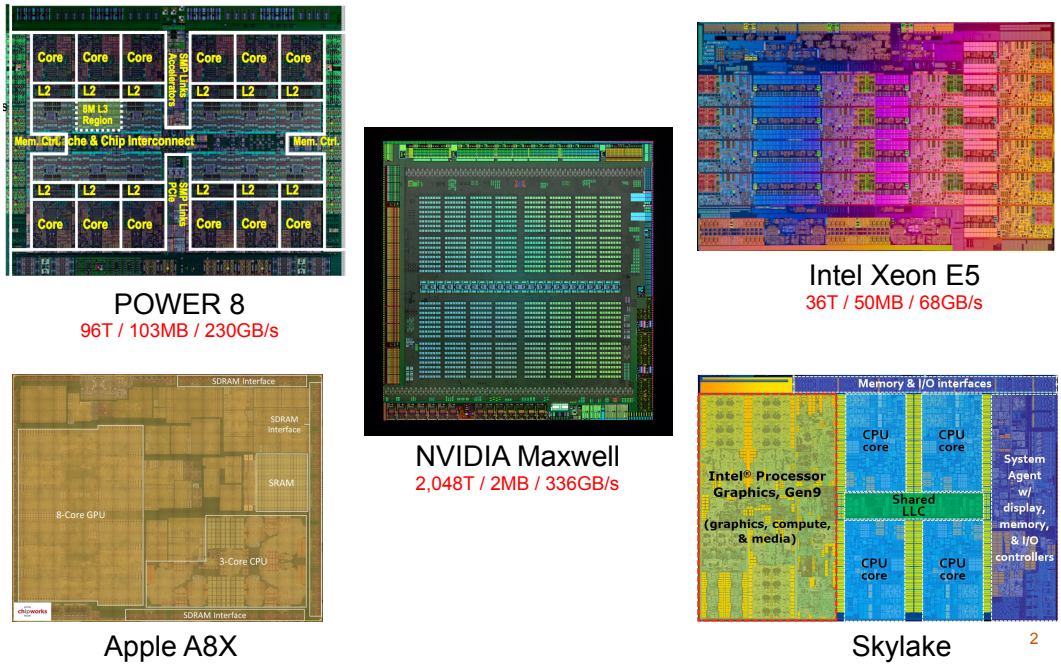


Figure 1.1: Widely used computer chips circa 2014.

The first widespread throughput architectures were Graphics Processing Units (GPUs). Generating real-time 3-D graphics has insatiable demands for floating

point performance and extremely high arithmetic intensity, meaning that very few instructions (˜2%) need to access memory. In order to get the most out of every execution cycle, GPUs employ fine-grain hardware multithreading. Every cycle, an instruction can be issued to the execution pipeline from a new thread with no performance penalty. To fit as many computation units as possible on chip and use as little power as possible, these units were not optimized for single threaded latency. As a result, it takes hundreds of cycles for a main memory request, which means hundreds of active threads are needed to keep the pipeline full in the presence of a memory access. Because this latency was even higher than the number of non-memory instructions per memory instruction in graphics programs, it meant that a lot of bandwidth was needed to support the memory needs of so many threads. GPUs employed a form of caching, albeit different from traditional CPUs. They employ a hierarchical memory system in which groups of 64 threads used scratch-pads to store local data, and clusters of these groups shared small L2 caches that were designed not to reduce instruction latency, but to amplify the effective off-chip bandwidth. As the thread count of GPUs continued increases, more recent GPUs have added L1 caches, and have begun to reduce latency to improve throughput performance rather than simply adding more threads.

Meanwhile, in the CPU world, several factors were combining to halt the annual increases in single thread performance. The end of Dennard scaling meant that transistors wouldn't necessarily run faster, or use less power than in prior generations. The two main architectural approaches to increasing single thread performance, deep pipelining and superscalar issue, had hit points of diminishing return. Most importantly, the increasing relative latency to main memory and a limitation to on-chip power meant that powerful cores were just stalling and waiting on memory accesses. As a result, computer architects could only increase total performance per chip by adding multiple cores, and multiple threads per core. Figure

1.1 shows that large scale on-chip parallelism is now ubiquitous. Now, both GPUs and CPUs have a hierarchy of on-chip memory and off-chip bandwidth shared by a large number of threads. What makes this different from traditional parallel programming is that threads must divide up available on-chip cache between them. This results in the average latency of a memory instruction being no longer fixed, but dependent on the number of threads sharing the cache. This can easily lead to a "downwards spiral" in performance. As threads are added to a system to cover the latency to main memory, cache hit rates drop, and the average latency to main memory actually increases, necessitating even more threads, which further increases latency.

The main thesis of this dissertation is that the characteristics that can lead to such a downwards spiral also provide a unique opportunity to improve throughput performance. Even very small changes to the efficiency of on-chip caching can mean very large improvement in throughput performance. In this dissertation, we derive a mathematical understanding of the non-linear relationship between active thread counts and throughput performance. We further assert that general optimizations for this special regime are possible, and that a key area for these optimizations is in the on-chip memory system. The goal of this dissertation is use this knowledge to derive cache optimizations in both hardware and software that boost the throughput performance of general purpose applications for both coarse-grained and fine-grained parallel architectures, and that require no explicit tuning or feedback mechanisms. We successfully derive and demonstrate two such general optimizations - one in software, the other in hardware. This is summarized in more detail in the following organization description.

1.1 Dissertation Organization

This Dissertation is organized into the following sections:

Immediately following this introduction, Chapter 2 describes the most important related work in each major subject area of this Dissertation.

Chapter 3 provides a mathematical derivation of throughput performance, to understand in greater detail the tradeoff between throughput performance and the number of threads in the presence of caching. In this chapter, we derive total chip performance as a function of an application's data reuse distribution, cache latencies, and thread count. This model demonstrates that in the design spectrum between coarse and fine-grained throughput architectures, there are actually narrow islands of high performance separated by oceans of poor performance, and these optimal points are not the ones commonly targeted. We show that the benefits of caching are quadratic in hit rate, so if an application has data reuse, caches can provide a tremendous performance benefit. However, this performance is very fragile — benefits drop off extremely quickly with hit rates (or increasing thread counts), at which point the system is quickly reduced to an off-chip streaming architecture. The results of this model make it clear that caches can potentially offer substantial benefits to throughput but must be highly optimized to ensure extremely high hit rates. However, optimizations in the software or hardware that either increase cache hit rates or reduce effective cache latency can make a tremendous improvement in throughput perfromance.

In Chapter 4, we apply the concepts introduced by the mathematical model to solve the current problem of intra-chip scalability of supercomputing applications. Despite being the most scalable parallel software known, these applications cannot scale across the individual cores within a single chip. We demonstrate that the bottleneck is indeed the shared memory system, and that these applications are exactly in the regime predicted by the model to be the best candidates for throughput optimization. We analyze several aspects of the classical software stack that are not optimized for throughput systems, and derive a compiler technique called "loop

microfission" that increases scalability across cores by up to 50%.

Chapter 5 shifts our focus from coarse-grained to fine-grained throughput architectures. In particular, we study an abstracted version of a NVIDIA style GPU called a Replay Architecture. We describe a custom simulator used to overcome the deficiencies of contemporary simulators that (1) did not model the on-chip memory system in sufficient detail to capture the most important effects, including the long latencies involved, and (2) were slow enough to make examining the large spatial and temporal scale needed for a meaningful study of large scale memory effects impractical. As a result, this is the first modern study to show that the high latency of the primary on-chip memory system of GPUs acts as a first order performance concern. We then perform preliminary characterizations of the Rodinia and Parboil benchmarks, discovering that they are actually amenable to caching and throughput performance improvements.

In Chapter 6, we look for an architectural enhancement that will boost the efficiency of the on-chip memory system, and hence throughput performance. In this case, we increase both cache hit rates and effective cache hit latency, both of which the model predicts to give considerable improvements to throughput performance. We find that for high memory intensity applications that have regular memory access patterns, the first order performance bottleneck is due to conflict misses in the cache system and bank conflicts in the on-chip caches and scratchpads. We derive Arbitrary Modulus Indexing, a technique to efficiently implement in hardware non-power-of-2 indexing, which can be leveraged anywhere on a chip. When applying AMI to the L1 cache banks, L1 sets, and scratchpad banks, we find that it completely eliminates bank and set conflicts (conflict misses), reducing average instruction latency and providing a significant performance boost in benchmarks with regular memory access patterns.

Finally, this dissertation concludes with Chapter 7 as a final summary of this

work.

1.2 Major Dissertation Contributions

Each study in this dissertation produced numerous insights and techniques, so we list only the most significant contributions here.

- Mathematical Derivation of Throughput Performance: We derive a model of throughput performance as a function of cache space per thread, providing more detailed insights as to the potential benefits and fragilities for on-chip memory systems as thread level parallelism increases. We show there is a non-linear tradeoff in throughput performance that is not well understood, and that optimal operating points are not ones commonly targeted. Most importantly, we show the sensitivity of performance to changes in thread counts, and that minor tweaks can have major effects. We demonstrate the importance of more efficient caches and algorithms are needed to achieve these potential benefits.

- Multicore Analysis: We demonstrate that for multicore applications, traditional performance metrics and sampling techniques are insufficient. A core issue is that memory disturbances from sampling can persist for billions of cycles, forcing sample intervals to become too large to capture effects from important, yet short lived functions. We developed a light weight, non-statistical, near zero memory disturbance library for capturing hardware performance counters during live program runs.

- Loop Fission: A general, localized compiler optimization that improves the intrachip scalability of multicore programs by reducing their working set size.

- Custom Throughput Simulator: We developed the first modern GPGPU simulator that provides a detailed model of the on-chip memory system and accurately captures the impacts of long latency pipelines typical of Replay Architectures. The simulator is 2-3 orders of magnitude faster than traditional simulators, allowing the analysis of benchmarks of sufficient size to get meaningful insights into the performance characteristics of the memory system.

- Benchmark Characterization: In studying benchmarks from the Rodinia and Parboil benchmark suite, we found counterintuitive results. Namely, that many of these applications had very high memory intensity, yet had memory access patterns that would be very amenable to caching. We further found that applications with "regular" memory access patterns were actually much more challenging for the memory system than those with "irregular" access patterns.

- Arbitrary Modulus Indexing: This dissertation introduces Arbitrary Modulus Indexing (AMI), a technique that allows hardware to efficiently access a non-power-of-2 number of items. We demonstrate the efficacy of applying AMI to the primary memory system of a Replay Architecture, where it eliminated bank and set conflicts in the L1 cache and scratchpad, improving throughput performance. However, this approach can easily be applied to other architectures, and has broad applications for solving a wide range of computer architecture pathologies well beyond the memory system.

Chapter 2

Related Work

The on-chip memory systems of coarse and fine grained throughput architectures is a broad subject that covers many areas of computer architecture. In this chapter, we group related work by general subject area.

2.1 Multicore scalability and metrology

The literature is full of work on optimizing specific scientific kernels for multicore CPUs, including stencil computations [17], sparse matrix-vector multiplication [82], and even a Lattice-Boltzmann computation [63]. Development of the STREAM benchmark suite [53, 54], which demonstrates highly optimized memory performance, was achieved based on detailed architectural knowledge. Mytkowicz et al. analyzed SPEC benchmarks using PAPI to read hardware counters and noted that measurement error due to measurement perturbation, OS context, and compiler flags can be so significant and unpredictable that correct performance conclusions cannot be easily drawn [59]. In fact, our study of a full-scale application in the context of a modern multicore supercomputer showed almost an order of magnitude greater perturbation. Their optimizations are measurement instead of code based –

they recommend running tests over a wide range of OS parameters and validating any performance conclusions by conducting detailed tests to confirm hypotheses. Our research goes into more depth about measurement errors and how to avoid them or at least limit their impact.

Dongarra et al. focus attention on the impact of multicore architectures on scientific applications [24]. For example, they observed that multiple cores on a chip cannot be treated as a traditional SMP due to shared on-chip resources, and, as a result, new scientific code would become much more complex because it would have to take increasingly varying architectures into account. Whereas they further identify potential difficulties in programming and compilation for multicore architectures, they do not focus on issues with performance measurement.

Jayaseelan et al. investigate various performance characteristics of three compilers on the integer SPEC benchmarks, focusing on PAPI hardware counters as primary metrics [38]. They observed that different compilers do better in different contexts, and that judging a compiler only by total program performance can be a mistake. They also note compiler differences in cache miss rates and fundamental instruction counts and recommend some optimizations. Our research focuses more on memory and less on CPU optimizations. We note the impact of the compiler on memory access patterns in the multicore regime and recommend fundamental but simple changes to the way compilers optimize code for multicore processors.

The previous research that most directly addresses bottleneck analysis and optimization approaches specific to multicore chips is the Roofline model from Berkeley [83]. This model defines a number of key performance metrics and uses microbenchmarks to estimate realistic values for a given hardware platform. Comparing the actual performance of several HPC proxy kernels (the original seven Berkeley dwarfs) then provides insight into multicore bottlenecks and code optimization techniques on a half dozen single-chip multicore systems. While insightful and useful,

the Roofline paper focuses on a much coarser level, examines a different scale of programs, and targets more traditional optimization techniques like blocking and CPU-centric approaches such as successful vectorization and balancing multiplies with adds. It does not employ hardware counters, nor does it focus on the difficulties in making accurate measurements or interpretations of those measurements. Roofline does not consider the on-chip memory hierarchy or the complications of DRAM pages and access patterns along with their performance impact on high-level compiler optimizations.

2.2 Mathematically Modeling Throughput Performance

Modeling Throughput Performance. The closest work relating to our mathematical model of throughput performance was [32, 33]. The authors also graphed the throughput performance function, $P_t(N_t) = N_t \times P_s(N_t)$, noting the two primary operating regimes and graphing sensitivities to varying parameters. In [33], they demonstrated that several benchmarks could be fit to the model, provided their kernels did not have phase changes. The primary differences were that the authors did not use any mathematical analysis that might have lead to further analysis, and a deeper understanding of performance or sensitivities. They also focused on a different domain, a CMP context using CMP benchmarks, missing key sensitivities when bandwidth is limited. Finally, they did not apply their model to the notion of general throughput performance or enhancing throughput architectures, but rather designed their model as a way to predict application performance.

2.3 On-chip Memory Systems for Throughput Architectures

Throughput Memory Systems. Rigel [42] designed an on-chip memory system

for a thousand core MIMD throughput architecture. To reduce average energy per instruction, each of 8 cores in a cluster had an extremely small instruction and data cache in the 1-2KB range, refilled from a 64KB cluster cache. The main issue addressed in the Rigel memory system was coherency — a combination of software load/store hints and a software runtime maintained coherence between tasks using a bulk synchronous programming model (BSP), in which writes during a task do not become globally visible until after a mass synchronization point. Special load/store commands were used to access global data that resided permanently in the L3, ensuring coherence. This dissertation completely ignores coherence issues, as most of the applications we examine do not require cache coherence across cores. However, modern GPUs do obey the BSP model with respect to task boundaries.

More recent articles on on-chip memory systems for throughput architectures focus on prefetching [49], or maximizing throughput or associativity [71]. We ignore prefetching, focusing on the regime where off-chip bandwidth is saturated, or where capacity misses are more important than compulsory misses. We believe our work on reducing conflict misses will be simpler and more effective than [71]. We additionally focus on the holistic view of the memory system, looking at the efficient interplay between thread scheduling and cache topology as well as cache index policies.

Thread Scheduling Policies. The most recent work in the throughput space explores the notion of utilizing two-level thread scheduling both to reduce power [28], and to avoid full task stalls on memory loads [60]. Thread throttling to increase performance has also been well explored with conventional CMPs, e.g., [15], but has been less studied in the context of throughput structures due to their historical low reliance on caching. In this work, we focus on indirect thread scheduling as a byproduct of caching, but find that the longer latencies to the primary memory system otfen require all threads to be active.

Cache Indexing Policies. There have been a number of articles on cache

indexing in the context of a conventional CPU and for virtual memory. The notion of prime banking has come up multiple times and has always been the best performing index scheme [44], but lacked efficient methods of implementation. In [85], the notion of a prime stride was efficiently implemented on vector supercomputers, providing a 3x performance increase, despite using an incorrect starting address. In [45], an efficient mechanism was discovered for prime banking, but the paper's focus was on other indexing schemes. We derive a similar efficient implementation scheme for prime banking in a different way, and avoid a costly correction step. Furthermore, we demonstrate the notion that odd-banking is a sufficient proxy for prime banking, as power-of-2 conflicts remain the primary issue for conflict misses. Several studies including our own have shown that conflict misses are primarily an issue when cache per thread is low — and we are the first to apply such techniques in a throughput architecture. Similar to the Z-Cache[71], we have evaluated the use of skewed associative caches and cuckoo hashing, but found them to have inferior efficiencies to prime banking. We also focus on policies which are closer to least frequently used caches and which do not cache every load.

Cache Replacement Policies. Belady[7] outlined the entire spectrum of placement methods along with their relative performance merits, without much exploration of detailed implementation issues. Belady showed that the best replacement policies were those that included information about a given address's previous use, and proposed an oracle algorithm, MIN, which used Farthest In Future Use (FIF) replacement as optimal. It was long recognized that the two main classes of algorithms were based on either recency, which is optimal for the stack depth distribution, or frequency, which is optimal for a distribution based on the Independence Reference Model. Frequency is a far better match for computation kernels in which some data is used more frequently than others, but Least Frequently Used (LFU) caches have two primary implementation issues — they are slow to react to change in

distributions, and implementing LFU can be complex. The Advanced Replacement Cache (ARC) [55] dynamically adapts between frequency [70] and recency models, keeping a limited tag history cache to leverage information about past use. Generational models [35] and Dynamic NUCA caches (D-NUCA) [46] also use a ranking with hysteresis to approximate frequency of use. However, most cache studies tend to use software replacement algorithms for slow virtual memory paging. However, hardware caches tend to be built on set-associative caches with ways far larger than sets. As a result, we found real cache behavior is actually most dominated by direct way mapping, rather than set replacement policies.

2.4 Arbitrary Modulus Indexing

As computers employ binary numbering schemes, hardware memory resources such as number of banks or sets are typically found in powers of 2. Likewise, software data structures are often accessed in a power-of-2 (or multiple thereof) stride as programmers optimize their algorithms for hardware implementations. Many have observed that mapping conflicts are worst when the index modulus and the memory access stride share a common divisor [85, 73, 76], so power-of-2 strides combined with power-of-2 address mapping schemes often lead to undesirable levels of resource conflicts. This problem has been well-studied in the context of interleaved memory bank and cache set conflicts, and both software and hardware approaches have been used to mitigate the conflicts.

2.4.1 Software Approaches

A programmer or compiler can carefully adjust the data layout of the code by padding array sizes or providing cyclic rotations of rows. Both library APIs [14] and language extensions [11] have been proposed to abstract array access and allow a programmer to independently specify the static memory layout of arrays. All

of these approaches impose a burden on the the programmer and are impossible in cases where the size or access patterns of the data are dynamic [16]. While hardware solutions have been proposed to ease efficient data layout [8], it is preferable to reduce conflicts without modifying the structure of the data.

2.4.2 Power-of-2 Indexing Algorithms

Bit hashing. While numerous index hashing techniques that take the form of simple bit operations have been proposed [72], such schemes do not prevent all power-of-2 self-conflicts and thus end up trading off improved performance on some access patterns with reduced performance on others [44, 45]. Application-specific hashing schemes devised offline [29] or dynamically [66] reduce conflict misses by up to 30%, but AMI removes nearly all conflicts, even with a single, static indexing scheme across the benchmarks we examined.

Reordering. Buffering and reordering strided memory accesses has been shown to reduce bank conflicts [25, 79] within single streams of access. Reordering can be combined with advanced hash techniques such as Galois Fields [69] to statistically reduce conflicts for a general mix of vectors. However, SIMD lanes do not have per lane buffering or reordering, and conflict rates as low as 3% can halve throughput.

Associativity. Adding associativity to a direct-mapped cache can reduce bank conflicts but is costly to implement past a small number of ways. More sophisticated approaches use multiple bit-hash schemes, either on multiple ways, such as skewed-associative caches [72], or on the same way, such as column-associative caches [2], hash-rehash caches [86], ZCaches [71, 26], or various indirect indexing schemes [35, 8]. Many of these schemes are difficult to implement in practice. Approaches that increase associativity are primarily limited to reducing set conflicts in caches, as opposed to banks or scratchpads. In most cases, such approaches are

orthogonal and complementary to non-power-of-2 caches, but have limited applicability to banks, scratchpads, small L1 caches, or modern GPUs. In addition, the parallel lookups required in associative caches increase power consumption; AMI can eliminate set conflicts without resorting to an associative cache.

2.4.3 Non-power-of-2 Indexing Algorithms

Prime number indexing: Prime moduli have long been assumed to be the best indexing choices to minimize conflicts since they have the fewest divisors. In some cases, prime bank indexing has proven to have sufficient advantages over simple power-of-2 mapping as to be worth implementing even as an expensive series of iterations [48].

As the latency and area required for index computation have become more critical, more efficient techniques for prime index computation have been sought [4]. The most significant optimizations in the prime index computation of general moduli involve the application of modulo/remainder algebraic properties, such as the Chinese Remainder Theorem (CRT) [27]. These approaches break an address into smaller chunks and apply a modulus operation by a linear combination of narrow address digits and constant weighting parameters. In the best case, when applying CRT to moduli of the form $2^N - 1$, the coefficients become 1 and the operation reduces to computing the modulus of a narrow sum of numbers [77].

Mersenne prime indexing. Given the relative efficiency of implementing moduli of the form $2^N - 1$ and the assumption that prime moduli are preferred [18], Mersenne primes of the form $2^N - 1$ have been the most efficiently implementable non-power-of-2 moduli studied. They also have the advantage over other primes of being close to powers of two and thus more easily integrated into hardware in which resources are sized in powers of 2 [44, 45]. However, Mersenne primes give few candidate moduli (the first 5 Mersenne primes are 3, 7, 31, 127 and 8191) and these

are spread so far apart that there is often no appropriately sized choice. On the other hand, non-Mersenne primes are more plentiful but farther from powers of 2 and thus harder to integrate into other hardware. No sufficiently efficient implementations of non-primes other than powers of 2 and those of the form $2^N - 1$ [77] and $2^N + 1$ [18] have been demonstrated, perhaps because they have been assumed to be likely to underperform prime moduli. Thus finding a non-power-of-2 modulus that is effective at minimizing conflicts and efficiently implementable for a given application may be difficult.

Chapter 3

A Mathematical Model of Throughput Performance

This dissertation defines a throughput architecture as one that must share on-chip resources in a way that most benefits total chip performance at the expense of single threaded performance. We further postulate that the most important on-chip shared resources are on-chip memory (caches or scratchpads) and off-chip bandwidth. But what, specifically is this tradeoff, and how does the distribution of shared on-chip memory and off-chip bandwidth effect total throughput performance and parallel efficiency? To better analyze and improve such throughput architectures, we must first define this fundamental architectural tradeoff in more detail.

In this chapter, we develop an ideal mathematical model of throughput performance. We can then use the mathematics to guide our intuitions and architecture design, express the optimum operating points for a given application, and express performance sensitivities to architectural and application characteristics. We keep the model as simple as possible to capture the desired behavior. This model also quantifies a number of performance limits — the maximum performance benefits of on-chip caching, the performance regime best suited to each application, the effects

of concentrated locality on performance, and most importantly, a deeper understanding of why this is the case.

There has been a gradual convergence over time of coarse-grained multicore processors and fine-grained throughput architectures, and many people have wondered what a final converged architecture might look like. Yet the current abundance of heterogenous chips perpetuates the either/or philosophy of throughput applications. The mathematical analysis below suggests reasons why this is the case, yet offers hope for an architecture that can accelerate both styles of applications — our notion of a "general purpose throughput architecture".

3.1 Defining Throughput Performance

To successfully optimize a throughput architecture, it is critical to specify precisely what we desire to optimize. The first quantitative steps in this direction were taken in designing the Rigel [40] and Tilera [81] architectures, which focused on designing cores that delivered the best performance for a given area. However, the amount of cache added was an afterthought and completely ad-hoc. We are now in an era where data movement costs more in terms of both power and latency than ALUs, and so a cache bank should be considered a first order performance unit. For example, a very small core with cache could potentially outperform a larger core with less cache in terms of both power and performance.

The goal of a throughput architecture, as described in section 4.2, is to amplify instructions per second by leveraging explicit task level parallelism to generate a very large number of active threads. In this case, total chip performance, P_T, is the product of average single thread performance, P_s, by the number of active threads, N_t. In the classic throughput model, unconstrained by off-chip bandwidth or on-chip context storage, maximum performance is obtained by simply increasing the number of hardware threads indefinitely. This is because in classical parallel

systems, each hardware thread is given additional hardware resources, namely, the node, and the only shared resource is the network, or to a lesser extent, shared off-chip memory within a node.

When there is a physical constraint due to sharing limited on-chip resources such as available bandwidth or cache, single thread performance, P_s, becomes a function of the number of threads, often rapidly decreasing when the number of active threads exceeds some threshold. The strength, or performance of a single thread, can be viewed as its computation rate, e.g. instructions per second (IPS) or cycle (IPC), or alternatively, instructions per joule (IPJ). Performance depends on the amount of independent instructions per thread (ILP) and the average latency per instruction, $L_{avg}(N_t)$ as:

$$P_T(N_t) = N_t \times P_s(N_t), \text{where } P_s(N_t) = \frac{ILP}{L_{avg}(N_t)} \tag{3.1}$$

Equation 3.1 illustrates that throughput performance is linear in the number of active threads and ILP per thread, but inversely proportional to average instruction latency. This statement can be viewed as analogous to Little's Law, an observation in simple queuing theory models of performance. In Little's Law, the average queuing length of a request is equal to the average arrival rate of requests times the average service time. In this case, the average queuing length can represent the needed task level parallelism (N_t) to balance the system, service time is the average instruction latency (L_{avg}), and arrival rate is total throughput performance, P_T. While such a model is interesting in its simplicity, the nonlinear relationship between single thread performance (service time) and the number of threads (queuing length) means that Little's Law is not effective in capturing the performance impact of parallelism when resources are shared between threads.

This model of throughput performance is highly extensible and can easily tell us anything we want. For example, one of the most important insights from the model lies in the derivatives of performance. For example, the points of maximum and minimum throughput performance occur where the derivative is zero, i.e.:

$$\frac{\partial}{\partial N_t} P_T(N_t) = P_s(N_t) + N_t \times \frac{\partial}{\partial N_t} P_s(N_t) = P_s(N_t) - N_t \times \text{ILP} \frac{\frac{\partial}{\partial N_t} L_{avg}(N_t)}{L_{avg}^2(N_t)} = 0. \tag{3.2}$$

At zero derivative,

$$P_s(N_t) = \frac{\text{ILP}}{L_{avg}(N_t)} = N_t \times \text{ILP} \frac{\frac{\partial}{\partial N_t} L_{avg}(N_t)}{L_{avg}^2(N_t)} \Longrightarrow 1 = N_t \times \frac{\frac{\partial}{\partial N_t} L_{avg}(N_t)}{L_{avg}(N_t)} \tag{3.3}$$

$$N_{t,opt} = -\frac{P_s(N_t)}{\frac{\partial}{\partial N_t} P_s N_t} = \frac{L_{avg}(N_t)}{\frac{\partial}{\partial N_t} L_{avg}(N_t)} \tag{3.4}$$

This can be shown to be mathematically equivalent to equal area efficiency:
$\Delta C/C = \frac{DelP_s}{P_s} = \frac{N_t}{Del_{N_t}}$, i.e.,
proportional change in cache space = proportional change in performance.
Let $\Delta N_t = 1$. Then,
$\frac{\partial}{\partial N_t} P_s = P_s \times N_t \Rightarrow \text{ILP} \frac{\frac{\partial}{\partial N_t} L_{avg}(N_t)}{L_{avg}^2(N_t)} = N_t \times \frac{ILP}{L_{avg}(N_t)} \Rightarrow N_t = \frac{L_{avg}(N_t)}{\frac{\partial}{\partial N_t} L_{avg}(N_t)}$.

Equation 3.2 illustrates that, if adding active threads has no effect on average latency, then each thread added increases throughput performance by the performance of one thread, which is why the classic multithreading recipe is to maximize the number of threads ($N_{t,opt} \to \infty$ as $\frac{\partial}{\partial N_t} L_{avg}(N_t) \to 0$). More generally, it shows that total throughput performance will then change according to the proportional change in latency caused by adding each thread, times the number of threads. This change can be large when either the number of already active threads is large, or when the change in latency due to adding a single thread is large. Finally, Equa-

tion 3.4 represents all the points at which the number of active threads yields local minimum or maximum throughput performance.

3.2 Arithmetic Intensity and Caching

Up to this point, no simplifications have been made - our model is exact. We will now simplify the model in describing the dependency of latency on the number of active threads. A large number of threads can stress on-chip resources in a number of ways, but negative performance feedback occurs when adding threads to mask latency actually increases average latencies, requiring more threads. There exists minor performance feedback baked into the core design: as more space is needed to store register contexts for each thread, accessing registers takes more energy and time, increasing register access latency. Having a very large amount of threads for each core complicates the circuitry to choose the next active thread to execute. Additionally, thread synchronization mechanisms may take longer with a large number of threads.

However, the largest latencies in the system are due to memory accesses. Requiring a larger number of threads to cover latencies increases the total memory footprint for a given kernel, leading to less efficient use of caches, less efficient use of bandwidth, and lower DRAM access speed As a result, average instruction latency can go up sharply as more threads are added. We will ignore the negative feedback effects of falling off-chip and DRAM bandwidth as random access increases, instead simplifying the mathematical model of $L_{avg}(N_t)$ to only capture latency added due to cache misses or bandwidth limitations. While the other feedback effects can be modeled as well, they do not qualitatively effect the results. Here is our latency model:

$$L_{avg}(N_t) = \hat{\mathcal{A}} \times L_{ALU} + \hat{\mathcal{M}} \times (L_{miss} - \Delta L_{cache} \times \hat{H}(N_t)),$$

where

$\hat{\mathcal{A}}$ = fraction of non-memory instructions (arithmetic intensity)

$\hat{\mathcal{M}}$ = fraction of memory instructions = $1 - \hat{\mathcal{A}}$, (memory intensity)

$\Delta L_{cache} = L_{miss} - L_{hit}$, the cache improvement in latency

$\hat{H}(N_t)$ = the application's hit rate for the cache

Hat accented terms represent canonical, non-dimensional values between 0 and 1, which we add for clarity. For example, dividing by L_{miss} as the characteristic latency yields:

$$\hat{L}_{avg}(N_t) = \hat{\mathcal{A}}\hat{L}_{ALU} + \hat{\mathcal{M}}(1 - \widehat{\Delta L}_{cache}\hat{H}(N_t)) = \hat{L}_{NC} - \hat{L}_C\hat{H}(N_t) \tag{3.5}$$

$$\frac{\partial}{\partial N_t}\hat{L}_{avg}(N_t) = -\hat{\mathcal{M}}\widehat{\Delta L}_{cache}\frac{\partial}{\partial N_t}\hat{H}(N_t) \tag{3.6}$$

Equation 3.6 describes the way in which caching can improve latency for a given application. This also shows the degree to which arithmetic intensity, $\hat{\mathcal{A}}$, diminishes the relative importance of caching. Assuming that $\hat{L}_{ALU}$ is approximately the same as $\hat{L}_{hit}$, then every non-memory instruction is as good as a cache hit. Another way to view this is that arithmetic intensity linearly reduces average latency in a manner similar to the hit rate of a cache. Because $\hat{\mathcal{A}}$ has such a profound impact on both throughput performance and the ability of caches to improve throughput performance, getting this parameter correct is critical. In our discussion on methodology, Section 4.3, we describe novel aspects of our infrastructure designed to correctly approximate $\hat{\mathcal{A}}$.

Finally, note that the ability of caches to help throughput also depends heavily on having a very low hit latency and a high miss latency. This equation demonstrates one characteristic that divides applications: to benefit from caching, an application should have a relatively low arithmetic intensity, as most HPC appli-

cations do, and have access patterns that can generate a good hit rate for relatively small caches.

As soon as we model the performance impact of caches, an unexpected characteristic of throughput performance emerges directly from Equation 3.1 as shown in Figure 3.1:

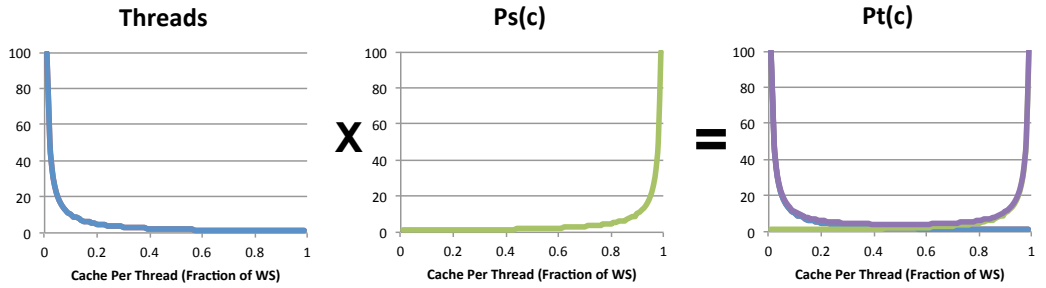


Figure 3.1: Once on-chip caching is modeled, throughput performance is maximum when either the number of threads is very low or very high, giving two possible optimum operating points. Y-axis are percent of maximum canonical value.

3.3 Hit Rates and Performance

How do we connect an application's data usage characteristics to a cache's hit rate and single thread performance? Fundamentally, applications are characterized by their working set size and the concentration of locality within that working set. Ideally, we'd like to cache those data items accessed the most times in a given interval. For the steady state kernels common in throughput applications, we can describe the importance of each piece of data by its frequency of access. Sorting all data items by access frequency results in a graph showing the concentration of locality in the applications working set, as shown in Figure 3.2. Assuming an ideal cache with Least Frequently Used replacement policies, the canonical integral of the access frequency graph yields the hit rate as a function of cache size, $\hat{H}(\hat{c})$.

An application's hit rate can be used to derive average instruction latency and performance, while the miss rate can derive unfiltered bandwidth out of the cache. We find most applications have a great deal of concentrated locality; however, in the worst case scenario, an application has no concentration of locality, access frequency is flat, and the canonical hit rate is linear, i.e., $\hat{H}(\hat{c}) = \hat{c}$. Once we define an applications data access characteristics, we can describe its optimum throughput performance.

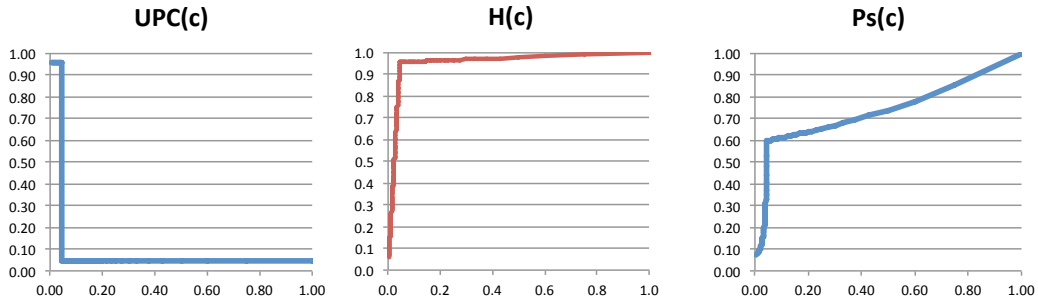


Figure 3.2: Converting application behavior to caching performance: Ideal hit rate is the integral of access frequency (concentration of locality). Performance is roughly the reciprocal of hit rate. Example is for a two level blocked algorithm with some data used 40x as frequently. Y-axis shows canonical values.

Note that the canonical cache space available per thread, $\hat{c}$, is inversely proportional to the number of threads. This means we are very quickly pushed into the far left side of the hit rate graph. For this reason, it is critical to design cache policies that have hit rates that degrade linearly with cache size. In comparison, an LRU cache operating on an application with no concentration of locality has the hit rate go immediately to zero when $\hat{c} < 1$.

Impact of reference frequency (Uses Per Cycle) on ideal cache performance: In the most conservative case of a flat profile (no locality), let w be the amount of working sets that fit on chip, i.e., total cache = wc and $\hat{c} = C/w$.

$$\hat{F}(\hat{c}) = 1/\hat{(c)} \Rightarrow \hat{H}(N_t > w) = \frac{w}{\hat{N_t}} \Rightarrow$$

$$\frac{\partial}{\partial N_t}\hat{H}(N_t) = -\frac{w}{N_t^2} \Rightarrow \frac{\partial}{\partial N_t}\hat{L}_{avg}(N_t) = \frac{w\hat{L}_g}{N_t^2} \tag{3.7}$$

We see that for the flat profile, the ideal cache hit rate (best possible case) falls quadratically with the number of threads.

To simplify notation, define $\hat{L}_{NoCache(NC)} = \hat{\mathcal{A}}\hat{L}_{ALU} + \hat{\mathcal{M}}$, and maximum cache latency gain $\hat{L}_g = \hat{\mathcal{M}}\widehat{\Delta L_{cache}}$ Then, for an application with flat access:

$\hat{L}_{avg}(N_t) = \hat{L}_{NC} + \frac{w\hat{L}_g}{N_t}$, and ...

$$P_T(N_t) = \frac{\text{ILP}}{L_{miss}} \times \frac{N_t^2}{N_t\hat{L}_{NC} + w\hat{L}_g} \tag{3.8}$$

Having chosen a mathematical description for the concentration of locality in the given application, we can now describe the complete throughput performance graph mathematically. But we still need to model the effects of saturating off-chip bandwidth.

3.4 Bandwidth Saturation

A real throughput chip has many architectural "caps", e.g., maximum possible FPC, maximum space for context storage, maximum buffers for instructions in flight, maximum socket power, maximum operating frequency, maximum space for caches, maximum available application parallelism and arithmetic intensity, and maximum bandwidth throughput the system. Because current off-chip bandwidth occurs at the perimeter of the chip, and process scaling continues to increase computation elements (area) at the expense of bandwidth (perimeter), the most prominent perfor-

mance bottleneck is the bandwidth wall. Simply put, we do not have the bandwidth to support the number of cores we can put on a chip, and bandwidth is currently the most expensive part of a chip's design. Reducing bandwidth requirements has a secondary benefit of addressing the socket power issue, as data movement and off chip access are an increasing percentage of total chip power.

When off-chip bandwidth is capped in a classical multithreaded processor, $P_T(N_t)$ becomes flat, therefore early advice to programmers was to run as many threads as possible. On modern, complex systems, once bandwidth is saturated, increasing the number of threads further hurts memory system performance by increasing both the working set footprint, which reduces caching benefits, and reducing locality and contiguity of accesses, which reduces effective off-chip bandwidth and DRAM responsiveness while increasing power consumption. The message is a simple one - at the very minimum, stop increasing the number of threads approximately when off-chip bandwidth is saturated. (We later show that it may be beneficial to stop much earlier.)

Bandwidth (words/second) is simply the amount of transferred data times its rate of transfer, or simply (words transferred per instruction) x (total instruction rate):

$$BW_T(N_t) = \{\mathcal{M} \times M(N_t)\} \times P_T(N_t) \Rightarrow P_T(N_t) = \frac{BW_{max}}{\mathcal{M}M(N_t)} \tag{3.9}$$

This is the equation for the final part of the performance graph, and we can see that as we increase threads past the point of bandwidth saturation, we get a quadratic drop in performance, as throughput performance not only is divided by an increasing miss rate, but the base throughput performance is based on single

thread performance, $P_S(N_t)$, which also falls with miss rate. In the case of highly concentrated locality, this will be the regime where $M(N_t)$ will be falling most rapidly. We can now illustrate the entire picture of throughput performance.

3.5 Throughput Performance

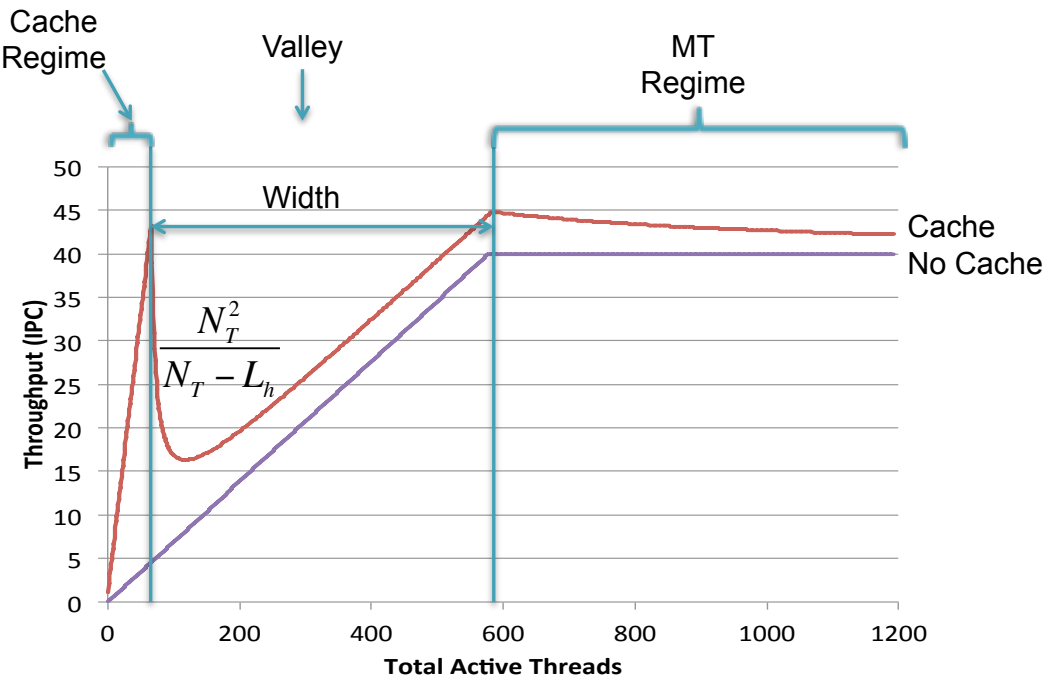


Figure 3.3: Annotated graph showing total throughput performance as a function of the number of threads, with and without caching.

We now have all the pieces to describe throughput performance as a function of the number of active threads, capped by a maximum bandwidth. A small number of parameters lead to the performance profile in Figure 3.3. The graph of $Pt(N_t) = N_t \times Ps(N_t)$ was first observed by Guz et al [33] in a more empirical fashion, but our mathematical derivation of the classic valley curve allows us to more deeply understand the effects of application cache response to direct archi-

tectural features. Additionally, our mathematical analysis describes performance implications and sensitivities as a function of application characteristics in much more detail, and we focus on performance regimes and benchmarks more suited to GPU-style throughput architectures than CMP architectures.

Given architecture parameters$\{L_{ALU}, L_{hit}, L_{miss}, BW_{max}\}$ and application characteristics $\{\mathcal{A}, \hat{H}(N_t)\}$, the throughput performance graph of Figure 3.3 has the following characteristics:

- For a given application and throughput architecture, there exists two potential operating points in terms of number of active threads that provide optimal performance. These tend to be sharp, well defined points, that would not be obvious to an expert programmer, but which can be reasonably approximated through direct hardware feedback, as defined by the model.

- There is a "cache regime" in which a very small number of threads can perform as well as a hundreds of conventional MT threads, their performance being amplified by the factor ($\hat{\mathcal{M}}\hat{H} \times \frac{L_{miss}}{L_{hit}}$). Deriving significant performance in the cache regime requires that an application has sufficient locality to get a high hit rate, that arithmetic intensity is relatively low, and that the cache hit latency is low.

- As total active threads increase, peak performance actually occurs a little beyond the point at which full working sets fit in cache. $\hat{N}_t$ is the canonical thread count, that is, the number of threads per amount of cache equal to the working set size of one thread. Peak single thread performance, P_s, would occur when $\hat{N}_t = 1$. For peak throughput performance, P_T, you want to increase the thread count, squeezing the amount of cache per thread to the point where misses occur, and P_s drops, such that the added thread count balances this drop and throughput performance is higher. To see this optimal

point intuitively, the more powerful the cache, the more expensive a cache miss is, and the more important cache hits are over total threads. If a cache miss will take forever, you want to fit the entire working set in cache. When the relative latency savings of the cache is lower (e.g. a high latency cache), the brute force of threads becomes more important, and optimum throughput performance occurs with less cache per thread, leading to a higher miss rate. However, in this scenario, the peak level of performance in the cache regime will also be considerably lower.

- The Valley: As total active threads increase beyond this peak, performance suddenly drops off sharply. This leads to a middle region, dubbed "the Valley" by Guz. In an idealized mathematical model, this drop off takes the form of a hyperbolic curve at an angle. The steepness of the drop again depends on the height of the fall and the strength of the cache. And this is an important observation wrt cache design. Because even when the hit rate drops just slightly below 100%, throughput performance drops off a cliff, quickly approaching the *minimum* throughput performance! The cache regime performance peak is so narrow and steep that in order to benefit from a caching system, we must enhance the efficiency of caches to ensure much higher hit rates on much smaller amounts of cache per thread.

  After hitting the bottom, hardware multithreading overcomes cache losses, and throughput performance increases linearly, as single thread performance approaches that of "no cache" levels. The end of the valley represents the point where traditional multithreading performance exceeds peak caching performance, which is roughly related to cache strength R. Note that Figure 3.3 is deliberately uncalibrated. For some applications, the peak performance in the Cache Regime will be much higher, while for other applications, the peak performance in the MT regime will be higher. It all comes down to application

data reuse and working set size.

- As the number of threads continue to increase, performance climbs again until reaching a peak just a little past the point where off-chip bandwidth is saturated, the MT performance peak. The model shows that even in this regime with low hit rates, caches still improve throughput performance, as throughput performance benefits quadratically with the hit rate when off-chip bandwidth is saturated. After the MT peak, performance again falls, as cache per thread continues to decrease.

This model has direct architectural implications. It shows the latencies most important to minimize and the cache regimes most important to optimize. It also provides guidance for automated feedback mechanisms that dynamically adjust the number of active threads to maximize throughput performance, by monitoring the arithmetic intensity of an application as well as its cache miss rates and off-chip bandwidth usage at runtime. Performance can be further enhanced by power gating off unneeded FPUs and using the power savings to overclock the smaller number of remaining FPUs.

Here are a some noteworthy performance sensitivities. The performance implications represent boosts to single thread performance, which in turn increases parallel efficiency, requiring less threads for the same performance, which can further boost single thread performance...

- Arithmetic Intensity, $\hat{\mathcal{A}}$: Linearly increases performance in the MT regime, with a quadratic benefit when off-chip bandwidth is saturated. Although it provides a small absolute performance benefit in the cache regime, it linearly diminishes the performance ratio between cache threads and MT threads, increasing the likelihood that the optimum point is the MT peak.

- Per thread ILP: Linearly increases performance in the MT regime. Can also

increase performance in the cache regime, or alternatively be expressed as TLP with increased cooperative threading.

- Degree of cooperative threading: Linearly increases performance in the cache regime, has no significant effects in the MT regime.

- L_{miss}: Reducing miss latency linearly increases performance in the MT regime. While having less direct impact in the cache regime, increasing miss latency linearly increases the ratio of thread performance in the cache regime relative to the MT regime, increasing the likelihood that the optimal point will be at the cache peak instead of the MT peak.

- L_{hit}: This has the sharpest impact on cache vs MT performance, because small changes in the hit latency create large changes in the miss to hit ratio and the hit to ALU latency ratios. Reducing the hit latency has the most dramatic increase on the relative performance of cache threads vs MT threads.

- $W, \hat{H}(\hat{c})$: Concentration of locality linearly increases the performance of Cache threads. It increases the performance of MT threads as well, but to a lesser degree.

3.6 Generalizing the Model

In exploring the mathematical tradeoffs of throughput architectures, this chapter has leveraged the example of reducing latency through cache hits. However, this model is of a more general nature — it is really about the relationship of latency to throughput performance. In this derivation, we looked at applications and cache heuristics that resulted in a roughly linear decrease in latency with a linear increase in cache per thread. That means that *ANY shared resource that results in a roughly linear decrease in latency will have an identical throughput performance curve! And*

even if the result is not linear, the shape of the performance graph is remarkably invariant to the actual latency tradeoff. This is important, because as we will see in this dissertation, this model correctly predicts throughput performance responses to any source of reduction in single threaded latency. In real systems, there are many other subtle effects, all of which can be modeled in a similar fashion.

For example, if groups of threads share data, it increases the power of caching. Effective bandwidth drops when multiple threads cause bus contention, DRAM page thrashing, or uneven use of off-chip DRAM channels. The amount of cache line coalescing effects the efficiency of both on-chip memory storage and off-chip bandwidth. The memory access patterns and working sets of applications can be altered by software or thread scheduling algorithms to change the point on the performance curve. More space can be devoted to tracking outstanding misses. There are also resource tradeoffs between multiple objects on the chip, for example, having multiple caches, or trading off cache performance with available bandwidth.

Any of these effects can be modeled, but the high level points remain: Reduce single threaded latency when possible so that you can reduce the amount of parallelism and increase the efficiency of the system. Because when you do increase this efficiency, even minor improvements may bring large performance benefits.

3.7 Summary

While existing studies had observed the existence of a "performance valley" in throughput architectures with a medium range of threads, this mathematical analysis provides a much deeper and more detailed insight into the benefit and sensitivities of this valley. First, we show that the variation in performance due to optimizations in the on-chip memory system is as significant as the ALUs or DRAM, so this area if worthy of analysis. We then demonstrate that the performance benefits of threads hitting in the on-chip cache is immense, amplifying per thread performance by the

ratio of off-chip to on-chip hit latency. Hitting this regime allows fewer threads and hence allocates more resources per thread, moving up the feedback spiral and boosting power efficiency in a super linear fashion. We further show that the points of optimal performance are not at the conventional locations of maximum cache hit rate or maximum bandwidth usage.

The details exposed in this analysis also illustrate the difficulties involved in obtaining this goal. The fall off in throughput performance with cache miss rates is mathematically sharp and inversely quadratic in the miss rate. To be an effective accelerator, caches must perform nearly perfectly. However, we also find a consolation prize at the other extreme: when cache hit rates are very low, they still provide a performance benefit quadratic in the hit rate. This is an example of a benefit of caching that is unique to throughput architectures, due to the tradeoff between threads and bandwidth. There are many ways to achieve this goal. In hardware, we can increase cache hit rate or improve (reduce) cache hit latency. In software, we can alter algorithms to have a smaller cache footprint or to change the pattern of memory reuse.

A final point is that this model is not simply about hit rates or even caches - it applies to almost any resource tradeoff in a throughput context. The overall message is clear. We cannot always choose an operating point of hardware threads or cache size, so we instead need to enhance the efficiency of caches, the efficiency of off-chip bandwidth, and even the memory access patterns of our applications. We need to look in general for any opportunity to reduce the latency of the memory system to improve throughput.

We will now apply this deeper understanding of the performance tradeoffs of latency, off-chip bandwidth, and throughput architectures. This dissertation will examine general techniques to reduce the latency of the shared on-chip memory system in both coarse-grained and fine-grained systems.

Chapter 4

Coarse-Grained Throughput Architectures

We now turn our attention to coarse grained (multicore) throughput architectures. Having discussed the tradeoffs that exist in Chapter 3, we attempt to discover how performance issues, analysis tools, and software optimizations differ for multicore systems than traditional unicore systems. How does the classic throughput tradeoff of single thread versus chip-wide performance effect software behavior? And what can be done to boost parallel efficiency? As always, the thesis of this dissertation is that we will find the performance issues in the shared on-chip memory system. In this study, the hardware is fixed, so we will examine software optimizations. While the resource contention issues are more complex than the idealized tradeoffs examined in Chapter 3, we show that the same concepts allow us to make a significant throughput improvement with just a minor increase in caching efficiency. In this chapter, we express throughput performance in terms of intrachip scalability, i.e.,

*Information in this chapter was previously published in the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2011 [19]. J. Diamond was lead author and performed all research, J. Brown supervised and funded this research, J. McCalpin contributed Top500 Supercomputer statistics, and S. Keckler, M. Burtscher and B.-D. Kim contributed to polishing the text.

the ability of traditional software to leverage multiple cores on the same chip.

4.1 Scalability of Supercomputing Applications

This chapter exposes a major problem facing the new era of multicore computer chips, or CMPs. To many people, a core is not much different than a processor, so taking advantage of multiple cores on a chip should be similar to classic multiprocessor algorithms. The most scalable algorithms known are supercomputing applications. This is by self selection — if an application cannot scale out to thousands or millions of cores, it cannot leverage the power of modern supercomputers. As a result, supercomputing applications tend to be embarrassingly parallel and have processes that only communicate or synchronize with each other during epoch boundaries. Supercomputer architectures went massively parallel much earlier than other computing markets due to their insatiable need for performance beyond what Moore's Law could provide for a single thread. Now that single thread performance has plateaued, maintaining the classic 1,000x increase in throughput performance per decade will require the number of cores used in a supercomputer to double almost every year.

Figure 4.1 shows that over time, the total cores per supercomputer have grown to the hundreds of thousands, and today growth is continuing into the millions. Historically, each time a new supercomputer arrived, scientists ran their existing code, and it simply worked - they got close to linear speedup via weak scaling. But Figure 4.2 shows a new trend: By 2007, the majority of supercomputers utilized sockets with multiple cores, and by 2009, the majority had 4 or more cores per socket. Today, it is not uncommon to see chips with 18-64 cores on them. Do supercomputing applications, known to be the most scalable, still scale as before? Figure 4.3 shows the surprising truth - that an archetypical supercomputing application with almost perfect weak scaling across a thousand chips, cannot weak scale

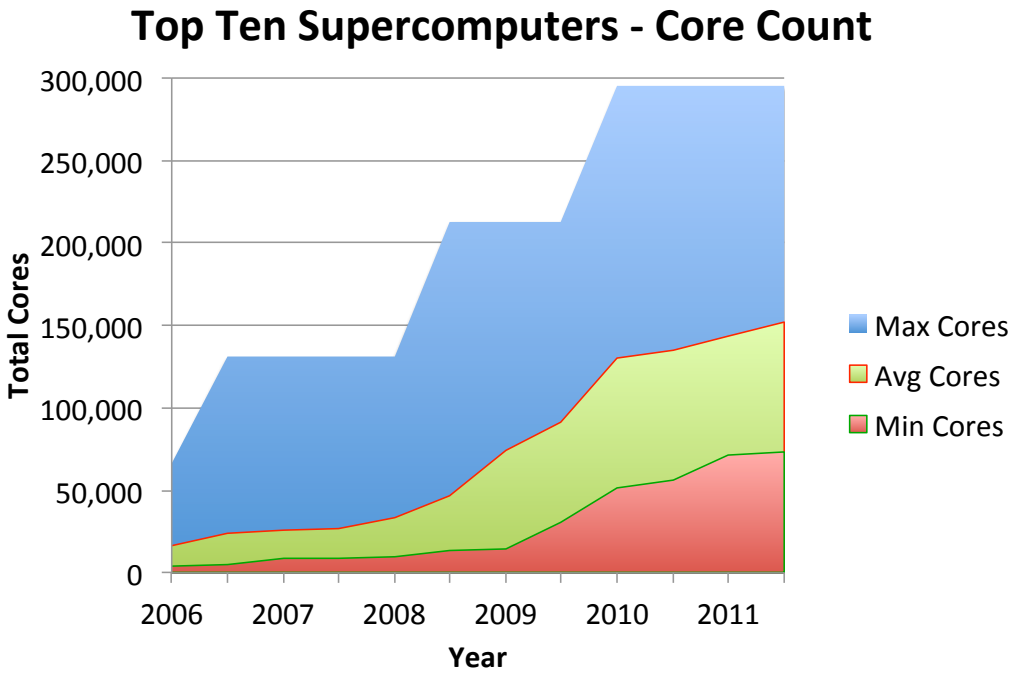


Figure 4.1: Total cores per supercomputer over time.

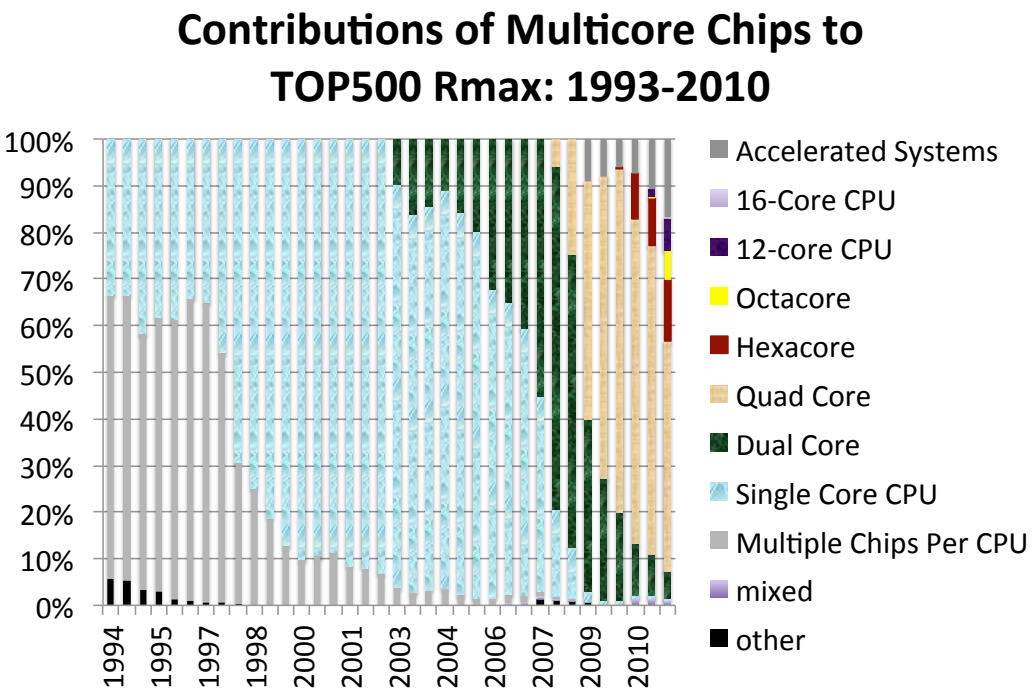


Figure 4.2: Cores per socket for TOP-500 Supercomputers over time.

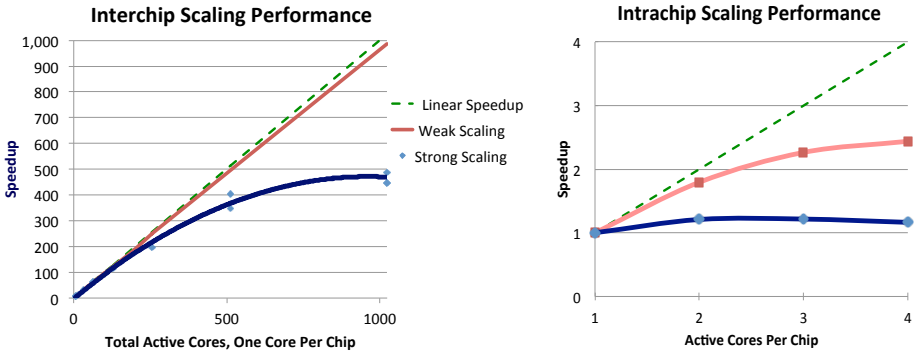


Figure 4.3: Interchip vs intrachip strong and weak scaling for a supercomputing application.

beyond two cores per chip!

The rest of this chapter goes on to show that scalability is not the only difference between multicore applications and multi-node applications. The optimization approaches are different from unicore architectures, the key performance metrics are different, the compiler algorithms are different, and the basic approaches to measurements and simulation are different as well. This chapter provides the new tools and approaches needed for multicore studies.

4.2 Outline of Study and Contributions

The computation nodes of modern supercomputers commonly consist of multiple multicore processors. To maximize the performance of such systems requires measurement, analysis, and optimization techniques that specifically target multicore environments. This chapter first examines traditional unicore metrics and demonstrates how they can be misleading in a multicore system. Second, it examines and characterizes performance bottlenecks specific to multicore-based systems. Third, it describes performance measurement challenges that arise in multicore systems and outlines methods for extracting sound measurements that lead to performance optimization opportunities. The measurement and analysis process is based on a case study of the HOMME atmospheric modeling benchmark code from NCAR running on supercomputers built upon AMD Barcelona and Intel Nehalem quad-core processors. Applying the multicore bottleneck analysis to HOMME led to multicore aware source-code optimizations that increased performance by up to 35%. While the case studies were carried out on multichip nodes of supercomputers using an HPC application as the target for optimization, the pitfalls identified and the insights obtained should apply to any system that is composed of multicore processors.

Although the compute nodes of modern servers and supercomputers are typically constructed with multicore processors, many legacy application codes have

been migrated to these architectures with little or no optimization for the multicore environment. As a result, these programs can often only use a fraction of the available cores effectively and may not obtain maximum performance on the cores they do utilize. While performance optimization for multicore chips is receiving intense attention, a systematic methodology is still lacking. Detailed measurement and analysis studies are scarce, even though it is well established that the analyses and optimizations for performance bottlenecks on unicore chips are not sufficient for multicore processors [24]. Multichip nodes, where the chips are multicore processors, add even more complexity to performance analysis through NUMA and interference effects. As a result, diagnosis of performance bottlenecks on multicore-based systems is difficult and laborious, as are multicore targeted optimizations.

Early source-code optimizations tended to be CPU centric. Such optimizations focus on reducing computation, by employing strategies of lowering branch costs through unrolling or memoizing previously computed values. Over time, the increasing importance of the memory latency and bandwidth made data-access issues dominate. Typical optimizations for unicore chips are core-local in scope and target better use of caching through blocking arrays and padding data structures. Other optimization techniques involve reading and writing data sequentially or with constant strides to aid prefetchers. Comparatively little work has been done on non-local memory optimization techniques.

Due to shared resources in the memory hierarchy, multicore applications tend to be limited by off-chip bandwidth. At first glance, other optimization strategies would seem moot as all applications would simply run at the speed of memory. However, we found that when the memory system is saturated, the order and pattern of data accesses becomes a performance-determining factor. We call this state of operation the *multicore regime*. As discussed in Section 4.4, the same optimization can yield very different results in unicore and multicore regimes. For example,

many optimizations that increase performance in the multicore regime can slow down uniprocessor code.

This Chapter describes the following contributions of this study:

- It examines traditional unicore metrics such as cache miss rates and demonstrates how they can be misleading in a multicore system.
- It presents an in-depth study of performance bottlenecks originating in multicore-based systems. The study identifies and characterizes three important bottlenecks (shared L3 cache capacity, shared off-chip memory bandwidth, and DRAM page conflicts) that are exacerbated by multicore chip architectures.
- It describes performance measurement challenges that arise in multicore systems including unpredictable and unrepeatable memory behavior, execution skew across cores, and measurements that disturb program behavior. It suggests remedies for each of these challenges and incorporates the these remedies into a systematic process for multicore specific performance measurement.
- It introduces a source-code optimization called loop microfission that is designed specifically to alleviate multicore-related performance bottlenecks. We observed a performance increase of up to 35% when applying microfission to a well-known supercomputing application.

We conducted this study using the HOMME atmospheric modeling benchmark code from NCAR, a complex application, running on supercomputers built upon AMD Barcelona and Intel Nehalem quad-core processors. While our studies were carried out on this single application, the process for measurement and optimization derived and the insights obtained apply broadly to systems comprising nodes of either single or multiple multicore chips that are used for executing computations that have predictable access patterns and good spatial locality.

The rest of the chapter is organized as follows. Section 4.3 describes our methodology. Section 4.4 introduces performance issues that arise in measurement and optimization of multicore chips and multichip nodes. Section 4.5 presents the optimizations and results. Section 4.6 provides conclusions and ideas for future work.

4.3 Methodology

The results reported in this paper are based on experimental measurements on two compute clusters: Ranger, a half petaflop AMD Barcelona-based supercomputer, and Longhorn, an Intel Nehalem-based supercomputer, both located at the Texas Advanced Computing Center. The HPC application we chose for our case studies is the High Order Method Modeling Environment (HOMME), developed by NCAR for their Climate Model 2 [34]. We used the performance tools gprof [30], mpiP [58], Pin [52], PAPI [64], perfctr [67], TAU [74], HPCToolkit [1] and PerfExpert [9] as well as the PGI and Intel compilers. This paper primarily presents our insights and results; additional data and details can be found in the accompanying technical report [21].

4.3.1 Systems

Ranger [68] consists of 3,936 16-way SMP compute nodes, each housing four 2.3 GHz AMD Barcelona-class Opteron quad-core processors for a total of 15,744 processors or 62,976 cores, and a theoretical peak performance of 579 teraflops. Each node has 32 GB of DRAM, of which each quad-core chip controls 8 GB. Nodes are connected by a 1 GByte/second InfiniBand network.

Longhorn [51] is a hybrid system with 256 nodes. Each node contains two quad-core Nehalem-EP processors operating at 2.5 GHz, between 48 GB and 144 GB of DRAM, and two NVIDIA Quadro FX 5800 GPUs. The nodes are connected

via a QDR InfiniBand interconnect. For this study, only the 2,048 Nehalem cores were used since our focus is on performance of homogeneous multicore nodes.

Ranger's Barcelona chips support almost 500 hardware performance counter events. Moreover, Ranger has a wide spectrum of performance tools installed, which is why we chose it as our primary host for measurements. However, the Barcelona chips have known core scalability issues [53], whereas the Nehalem chips were designed to overcome these issues. Hence, we chose Longhorn as the secondary host to ensure that our methods, analyses, and insights extend to other processor architectures.

4.3.2 HOMME Benchmark

Many HPC applications can be categorized by three general computational patterns. *Regular parallel applications* have patterns of computations and memory references that tend to be predictable. Examples include finite difference PDE solvers, dense linear algebra solvers, and stencil codes. *Irregular parallel applications* have dynamically changing data structures, such as adaptive meshes, to handle levels of detail or time-varying geometry, but may still maintain reasonable locality through strategies such as irregular blocking. *Graph parallel applications* have their computation time dominated by graph traversal and no spatial locality is guaranteed. Examples arise in bioinformatics and the intelligence community. While load balancing and communication are classical performance bottlenecks for irregular and graph-based applications, as will be shown later, we are interested in key intranode scaling issues that are readily present in all three categories of parallel applications.

For this study, we chose to employ a single large-scale benchmark rather than study a suite of small-scale kernels. HOMME contains dozens of functions and a wide spectrum of loop structures implementing numerous algorithms. Therefore, we believe that the measurements and analyses executed on HOMME span a sub-

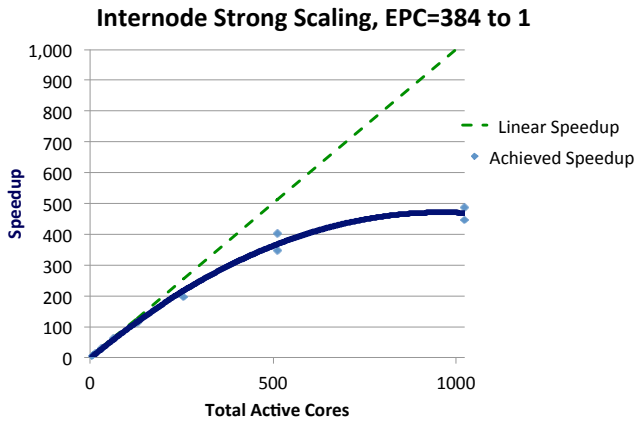


Figure 4.4: With fixed problem size, HOMME's node count can be increased 900 fold before the computation efficiency halves. The relative cost of communication increases linearly. Elements Per Core (EPC) describes the work done per core.

stantial fraction of the interesting space for regular HPC applications. HOMME is widely used by the supercomputing community and is one of the five HPC Challenge benchmarks [61] that are required for supercomputer acceptance testing.

HOMME (High Order Method Modeling Environment) [34] is an atmospheric general circulation model (AGCM) that provides 3D atmospheric simulation similar to the Community Atmospheric Model (CAM). The code consists of two components: a dynamic core with a hydrostatic equation solver and a physical process module coupled with sub-grid scale models. HOMME is based on 2D spectral elements in curvilinear coordinates on a cubed sphere combined with a second-order finite difference scheme for the vertical discretization and advection. It is written in Fortran 95 and is parallelized with both MPI and OpenMP. However, we are using the benchmark version of HOMME, which uses only MPI.

4.3.3 HOMME Scaling

HOMME is a very sophisticated and diverse example of a "regular" HPC application. It has been designed to scale well to tens of thousands of nodes and is highly

optimized for computation and locality. In fact, it exhibits near perfect weak internode scaling (computation per core is constant) to tens of thousands of cores, and excellent strong scaling (total computation is constant), requiring a 900-fold increase in node count before the efficiency drops to half.[†] Despite all these advantages, HOMME's intrachip/intranode scalability, both weak and strong, can be three orders of magnitude lower than its internode scalability.

HOMME is only locally regular. The cube mapped sphere is specified at a certain resolution, and that grid is then broken into elements consisting of 8x8x96 regular meshes. Each element stores physical flow properties as a structure of arrays, requiring 48 KB per value that a given function accesses and up to 9 MB per element. A distributed, general graph connects each element with its four neighbors. When running, each active core is assigned a list of elements to process. Each core computes physical quantities for each element, then in a communication step exchanges boundary data with four neighboring elements. The available memory per compute node limits the possible amount of work per core from a single element up to 801 elements. We use the metric elements per core (EPC) to indicate the amount of work each core performs for a particular mapping of the application to the available cores.

To examine the strong internode scalability of HOMME, we keep the problem size fixed at the standardized "large" resolution of 1,536 elements, and then vary the number of active Ranger cores from one to 1024. We use 4 cores per node (1 core per quad-core chip) to isolate inter-node from intra-node scaling effects. The solid line in Figure 4.4 shows that, as the core count is increased and work per core drops, the efficiency (performance relative to perfect linear speedup) drops relatively slowly. The ratio of communication time to computation time increases linearly as work is spread across more cores. That the efficiency drops slowly as

[†]If p is the number of cores and n is the total size of the data, "strong scaling" means $n(p) = n$ and "weak scaling" means $n(p) = np$.

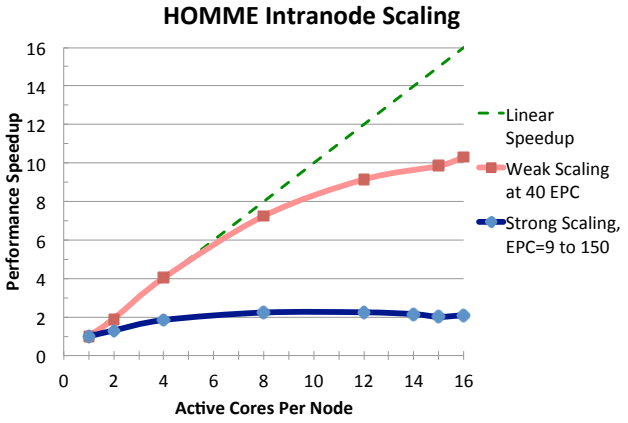


Figure 4.5: With intranode scaling, HOMME's efficiency falls below 15% when all 16 cores are used on a fixed problem size. Even weak scaling performance still drops by about 40%.

the communication overhead increases demonstrates the excellent strong scaling characteristics of HOMME across multiple nodes.

To examine weak scaling, we chose an input data size as close as possible to the standardized "standard" workload of 54 elements per core. Loads other than 54 elements per core were sometimes necessary to ensure that the total number of elements is divisible by the number of cores, removing issues of load balancing and interference. We found that HOMME's internode weak scaling is ideal, never falling below 94% of linear speedup. This scalability is a result of using only nearest neighbor communication.

For intranode strong scaling, we chose a fixed resolution that roughly centered on the standard work load of 54 elements/core. For weak scaling, we chose a constant 40 elements/core, the closest we could get to 54 elements/core without imbalancing the load. Sensitivity tests showed that intranode scaling properties were strongly independent on the load per core.

Figure 4.5 shows the strong and weak intra-node scaling profile from 1 to 16 cores in a four-chip node. For weak intranode scaling, the speedup at 4 cores per

chip is 60% of linear and falls off rapidly as more cores are used. For strong intranode scaling, performance increases very slowly when increasing cores per node, and provides little performance benefit beyond 4 cores per node. Regardless of the scaling regime (weak versus strong), improving single-node multicore performance would provide a substantial benefit to the performance of the application, independent of the inter-node scaling characteristics.

Our primary motivation in this paper is to investigate causes that prevent HPC applications with excellent conventional scaling properties from being able to utilize entire multicore supercomputers. If HPC applications cannot make good use of quad core chips, how will they make use of 30, 45, or even 64 cores per chip in the future? However, we will show that the multicore performance of HOMME lies directly on the right edge of the cache regime of the valley graph, and for this reason, even minor improvements to cache efficiency will have a signifiant benefit for performance, effectively increasing scaling.

4.4 Multicore Performance Analysis

This section explores the ways that conventional performance bottlenecks, metrics, and measurement error may be qualitatively and quantitatively different for multicore processors. Section 4.4.1 examines the basics of good multicore performance and why classical unicore performance metrics may be misleading. Section 4.4.2 describes three fundamental architectural bottlenecks unique to multicore chips that are the primary barriers to intranode scaling, new performance metrics needed to classify each bottleneck, and some common code styles that may exacerbate them. Section 4.4.3 describes how measurement artifacts change with multicore chips, providing a detailed categorization of memory measurement issues, the impact of *core skew*, and the increased importance of lightweight measurements. The final section summarizes a systematic approach to multicore performance analysis.

4.4.1 Multicore Performance Metrics

This section introduces a fundamental approach to measuring multicore performance along with comments about general performance metrics. It then demonstrates why traditional unicore performance metrics, such as L1 and L2 cache miss rates, do not adequately capture multicore memory issues.

Isolating intranode scaling effects

To simplify our experimental search space, we first determined how small a subset of Ranger we could employ to study intra-node scaling. We varied both the number of nodes and the number of cores per node, and found that the efficiency (performance as a function of total cores) was largely independent of the number of nodes and extremely dependent on cores per node. This feature is a testament to the internode scalability of HOMME and enables studying intranode scaling using as few as four 4-socket nodes. For the rest of the paper, we employ a total of 16 threads and spread them across 1 to 4 nodes, keeping the total work constant. We use a *density scaling* metric in which *core density* is defined as the number of cores used per active socket; core density varies from 1 to 4 in our experiments. We then used gprof to determine which of HOMME's procedures contribute most to the execution time and which of these functions suffer most from intrachip scaling issues.

Metrics for "good" performance

The performance of each function of an HPC application can be characterized by three rate metrics: Flops Per Cycle (FPC), the rate of algebraic computation; Instructions Per Cycle (IPC), how hard the CPU is working; and Loads Per Cycle (LPC), how hard the memory system is working. If any of these metrics approach the expected maximum for that architecture, then performance is good and further improvement must be accomplished through algorithmic optimizations. Advertised

peak rates for performance metrics in most chips are well-known to be unrealistic. However, reasonably attainable peak rate for application programs can be derived from targeted micro-benchmarks written in high-level languages; these empirically measured rates should be used for assessing performance of application codes. Note that the ratio of these three metrics to each other is fixed by the dynamic instruction mix, so generally only one of them can reach the hardware's maximum. If all three metrics are sub par, then in most cases the performance bottleneck is in the memory system.

Figure 4.6 shows these three metrics on Ranger for the most important functions in HOMME. For most programs, a combination of a high IPC, a high FPC and a high LPC indicate good performance. An IPC value of 2 is very good for Barcelona chips. In the middle of the graph are three functions with dramatically higher values on all performance metrics that are CPU-bound. The highest performance levels reached by these three CPU-bound functions is roughly 1.8 IPC, 1.3 FPC, and 0.5 LPC. These functions also simultaneously achieve the highest memory performance (LPC) as well. The remaining eight functions are memory bound, with an FPC of 0.1 or less and similarly low IPC and LPC. The multicore optimizations described in Section 4.5, which are aimed at improving the LPC, increase the FPC value to 0.25 FPC, still far below the reasonably attainable peak rate but more than a factor of two over the FPC of the original code.

Traditional unicore metrics are insufficient

In traditional performance analysis, L1 and L2 cache miss ratios typically provide insight into suspected memory bottlenecks. Furthermore, computation optimizations are often assumed to be irrelevant to performance, since the processor is waiting idly for data. However, we found that on modern multicore systems, these *core local* performance metrics were not good indicators of multicore performance issues for

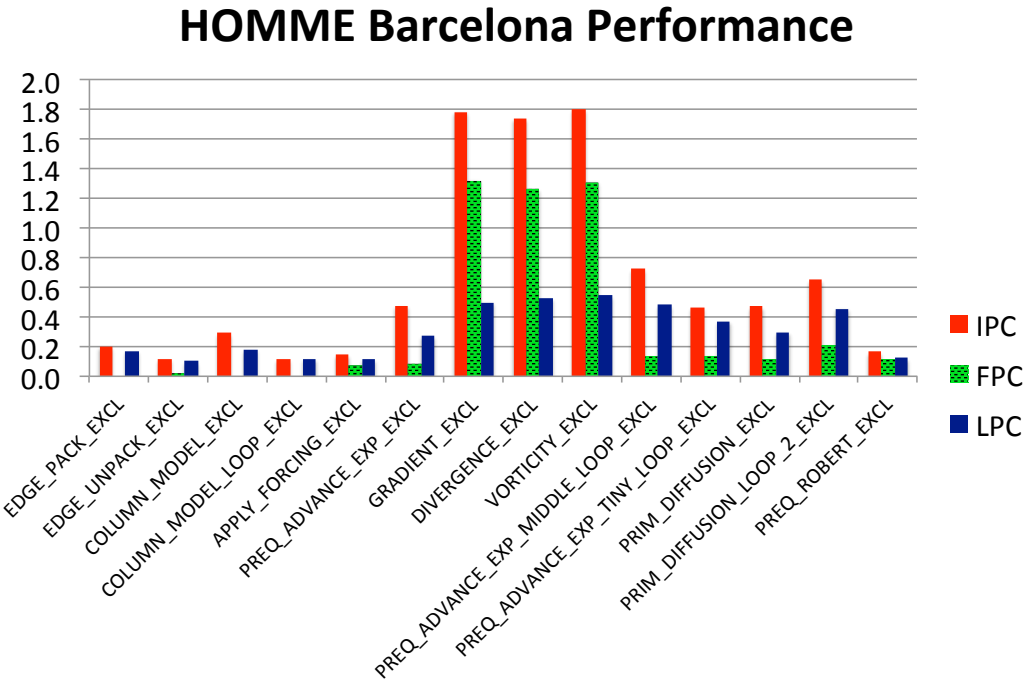


Figure 4.6: Performance metrics for HOMME's major functions.

three reasons.

L1 miss ratios may be misleading: A low L1 cache miss ratio by itself is not indicative of good memory performance on many chips because of the way in which hits are counted. When a load accesses a missing cache line, it counts as a miss, but all subsequent loads from that same cache line (which typically holds eight double-precision floating-point values) count as a hit, even before the hardware has been able to bring the cache line into the cache. By itself, this way of counting lowers the effective miss ratio to 12.5%, assuming eight values per cache line, for a regular stride-one application with all misses. In the presence of hardware prefetches, all accesses to a cache line that is not yet in the cache may be counted as hits, resulting in no L1 misses being recorded at all. Finally, once the (relatively small) maximum number of outstanding loads supported by the CPU has been reached, all further loads are stalled. The subsequent delay can be at least as long as a cache miss without being counted as one. As a consequence, a modern application can have great L1 performance but severe bottlenecks lower in the memory hierarchy. In fact, prefetching makes it fairly common to see functions with nearly 100% L1 cache hits actually fetching every value from DRAM and running at DRAM throughput rates. Due to bandwidth limitations of current CPUs, such direct streaming algorithms often suffer a 10-fold slowdown relative to the expected performance.

L2 cache miss ratios may not be predictive: We initially tried to infer memory bottlenecks by looking at the way the core density affected the L2 miss ratios. We assumed that if the L2 miss ratio was low or did not vary much with density, the system did not likely suffer from significant L3 and DRAM scalability problems. This assumption turned out to be false for functions with poor intrachip scalability, but could be corrected by looking at a few additional metrics such as L3 hit and miss ratios and DRAM page hit ratios. We found that the actual L2 traffic can be significantly higher due to prefetching, load replays, and coherence snoops,

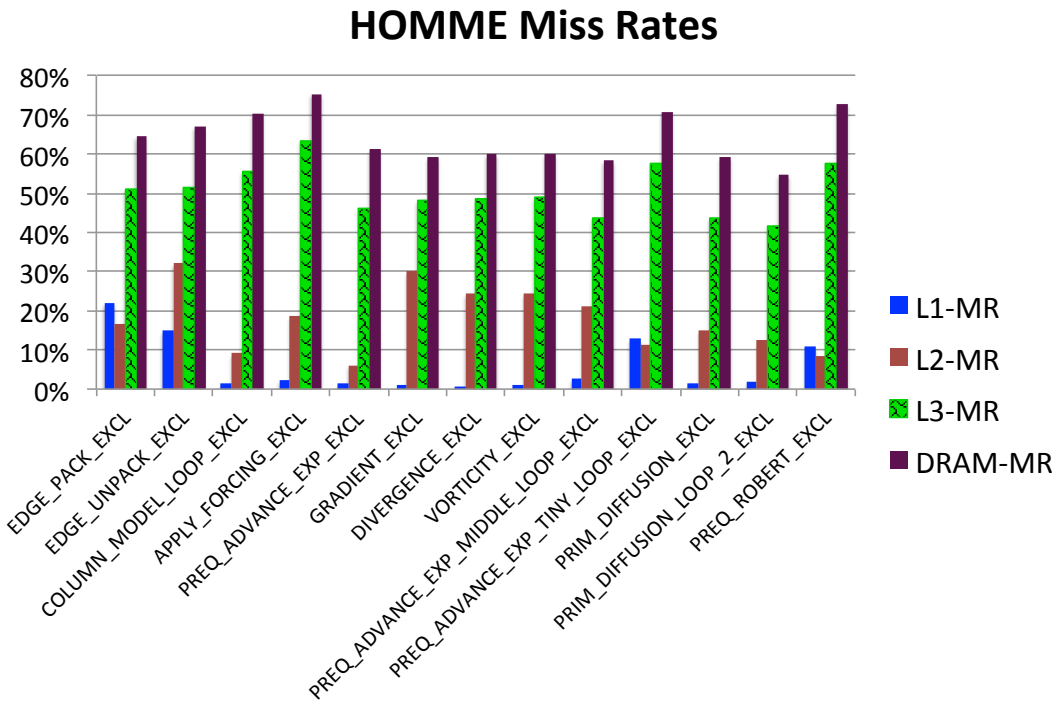


Figure 4.7: Miss ratio variability of major HOMME functions across the entire memory hierarchy.

which we observed to account for as much as 80% of the L2 bandwidth, even in pure MPI codes without any direct data sharing between processes. Figure 4.7 illustrates the miss rates at all levels of the memory hierarchy for the most important functions in HOMME. Correlating the heights of the bars reveals two important multicore effects: (1) L3 and DRAM miss rates are much higher than L1 and L2 miss rates and (2) the degree of L1 and L2 miss rates is not predictive of L3 or DRAM miss rates, and in fact is often uncorrelated with intrachip scalability. Miss rates increase further down the memory hierarchy, since locality is being skimmed by the higher caches.

Conventional CPU optimizations may be counterproductive: Applications are typically memory bound when multicore performance problems are present. However, CPU metrics and optimizations are still relevant because in the *multicore regime*, the exact order of load instructions has a first-class effect on performance. As such, many CPU unicore optimizations actually have the potential to hurt performance in the multicore regime. Figure 4.8 shows compiler flag effects on a baseline and aggressive multicore fission and blocking optimization on PreqRobert, as described in Section 4.5. Aggressive compiler optimizations such as "-O3" nearly doubled performance in the unicore regime of 1 core per chip, but the combination of optimizations drastically reduced performance in the multicore regime at 4 cores per chip by creating irregular access patterns. This example quantifies the degree to which CPU optimizations may behave differently depending on core density. The degraded performance stemming from the irregular access patterns generated by higher levels of unicore compiler optimization is why HPC users commonly use the -O2 instead of the -O3 optimization level.

In our experiments with the two compilers available on Ranger (PGI and Intel's icc 10.1), we also found that different compilers produce code with very different memory access patterns from the same source code. Specifically, we observed that

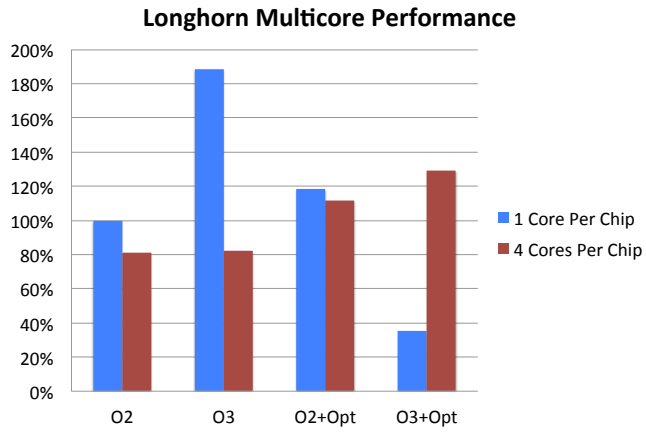


Figure 4.8: Effect of compiler optimization flags on Longhorn. "Opt" is a multicore optimization involving microfission and blocking arrays for the L1 cache.

the performance metrics for individual functions may vary by up to 40% between compilers, and metrics like cache and DRAM miss rates can vary by 3x.

4.4.2 Multicore Bottlenecks

This section describes three main architectural resources that we found to cause scalability issues in multicore chips: L3 cache capacity, off-chip bandwidth, and DRAM banks. This section also discusses how to measure and differentiate the effects of each of these bottlenecks, an important process as the optimizations for each bottleneck can differ.

Generally, issues in the memory system flow downwards: L3 capacity issues increase off-chip bandwidth demands, and off-chip bandwidth may exacerbate DRAM bank misses. However, it is still possible to experience any of these issues as the primary bottleneck.

L3 cache capacity

As seen in the previous section, the causes of poor intrachip scaling cannot be resolved from only L1 and L2 miss ratios. Further complicating measurements, the performance counters needed for L3 measurements and beyond are typically not supported directly in general purpose performance tools such as PAPI. We were able to use native hardware counters as defined in processor user guides [3, 36]. Fortunately, all of the tools we used for our case studies support passing native counter IDs to the hardware.

Diagnosis: If the L3 miss ratio significantly increases when going from the minimum to the maximum core density, it is likely that at least part of the performance problem is the L3 capacity, since each core can now only use a fraction of the shared cache. Note that the effective cache capacity can further be reduced by associativity issues and false sharing between processes. Since L3 cache misses directly translate into off-chip (i.e., node-level) bandwidth, applications with L3 capacity issues also tend to have memory bandwidth issues. Common solutions involve reducing a function's cache footprint, either in absolute terms or ephemerally, by using data serially. Some common pitfalls are listed next.

Storing intermediate values to reduce computation. When cache capacity is an issue, a good optimization is to reduce the memory footprint of a function by reducing or reusing temporary variables and redundant arrays. DRAM streaming speeds are 10 to 20 times slower than normal instruction execution, leaving ample time to compute values redundantly instead of storing them. Interchanging or promoting loops to minimize passes over the data can boost arithmetic intensity. Although these types of optimizations are worthwhile, they may be infeasible for large programs or when a programmer has limited familiarity with the code.

Operating on many values at once, instead of one at a time. An even easier optimization which does not require knowledge of the code is described

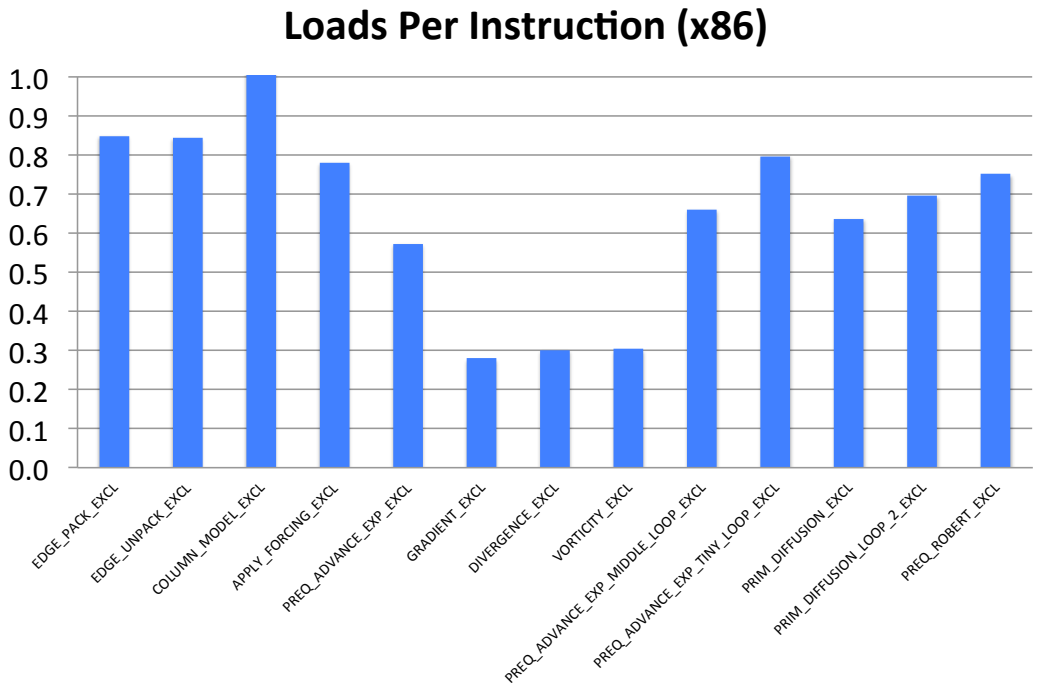


Figure 4.9: Loads Per Instruction: HOMME has an arithmetic intensity typical of HPC programs, with 60%-100% of the instructions accessing memory in 8 of the 11 most important functions.

in Section 4.5, in which a technique we call loop *microfission* results in drastically reducing the short term memory footprint of an application without effecting the algorithm.

Off-chip bandwidth

Off-chip bandwidth is now recognized as a first order bottleneck with multicore chips. Figure 4.9 illustrates LPI, or Loads Per Instruction, for the key functions in HOMME. More than half of the key functions require a memory access with almost every instruction executed. Because the x86 instruction set allows most instructions to include a memory operand, the LPI metric can effectively double as compared to a load/store architecture. HOMME is typical of HPC applications, which commonly

have little reuse of data values and therefore do not benefit much from a cache hierarchy. The off-chip memory accesses can be estimated by the L3 misses or DRAM data accesses. It can also be instructive to look at a function's "taper", defined as the number of bytes a given function needs to read per instruction at a given level of the memory hierarchy, typically off-chip DRAM. Most studies focus on bytes read from off-chip DRAM per instruction. Many functions in HOMME require one in three memory accesses to go off-chip, which can be used to determine the suitability of a given supercomputer to run a given HPC application.

Averages can be misleading. The performance weighted average for HOMME is just 0.8 bytes/flop off chip, but this average includes a significant number of functions that have no off-chip traffic. Peak values are critical to maintaining average performance, and we therefore recommend studying functions individually.

Determining a function's off-chip bandwidth needs requires minimum thread density. The bandwidth requirements of a function can only be judged when it is running unconstrained at one process per chip. If examined at full core density, it may paradoxically appear that the function's bandwidth usage has decreased, yet performance is worse.

Diagnosis: The magnitude of bandwidth bottlenecks, if any, can be gauged by the amount the function exceeds its share of off-chip bandwidth when running at 1 core per chip, which is the total off-chip bandwidth divided by the number of cores per chip.

DRAM pages

Perhaps the least studied multicore performance bottleneck involves contention over DRAM pages (not to be confused with OS pages.) A DRAM page represents a row of data that has been read from a DRAM bank and is cached within the DRAM for faster access. DRAM pages can be large, 32 KB in the case of Ranger, and there

are typically two per DIMM. This means that a Ranger node shares 32 different 32 KB pages among 16 cores on four chips, yielding a megabyte of SRAM cache in the main memory.

Diagnosis: Most systems have straightforward counters to estimate the total number of DRAM accesses and DRAM page hits, which may be significantly greater than the number of L3 misses due to prefetched data. The performance effect of DRAM page misses has become a first order concern in the multicore regime due to the following reasons.

DRAM contention can significantly impact performance. When a DRAM request is outside an open page, the page must first be written back to DRAM (closed) and the next page read out (opened), which adds 30 ns (69 cycles at 2.3 GHz) to the access time and reduces DRAM performance. As an optimization, the DRAM controller will attempt to close a row (write back the results) after a certain amount of time so the next access only needs to open a row, saving 15 ns. While this represents only 20% of the access time on Barcelona chips, newer processors have been reducing absolute latency to DRAM, so the importance of page misses will grow with time. As the number of cores per node grows, contention will increase, and making effective use of DRAM pages will be more difficult. More significant than this individual increase in latency is the reduction in effective DRAM bandwidth when DRAM bank miss rates are high. It is not uncommon to have a more than 50% DRAM bank miss rate when multiple cores contend for banks. While DRAM controllers have buffers and schedulers to mitigate these issues, they tend to by myopic due to limited buffer space.

DRAM Optimization is not necessarily complex. Many papers in the literature have focused on optimizing memory controllers and thread schedulers to reduce DRAM page contention, but few have explored the potential of employing high-level code transformations to reduce DRAM conflicts. Managing DRAM local-

ity at the program source level is difficult because we cannot easily control where memory is physically allocated, where those physical pages reside in the node, or when exactly the cores on a node access different pages. However, high-level code transformations can alter the locality in the access patterns to reduce the average number of conflicts and thereby increase performance. Section 4.5 illustrates the most important multicore optimization we found, using specially targeted loop fission to reduce DRAM page conflicts.

4.4.3 Multicore Measurement Issues

The previous subsections described *what* is important to measure. However, these new scaling and optimization behaviors resulting from multicore architectural bottlenecks also lead to performance *measurement* issues that are not merely different, but that can be extremely difficult to overcome. This section categorizes the ways in which multicore measurement issues fundamentally differ from traditional performance measurements, and follows with a description of the disturbance impact of *core skew* and techniques required to minimize measurement disturbances.

Classical optimizations focus on CPU performance and CPU monitoring. Their primary metric is speed, and aside from uncertainty regarding out-of-order execution, timing code could simply subtract out its own execution time and have minimal disturbance on running code. Although memory performance is critical for a wide span of applications ranging from HPC to databases that tend to have low arithmetic intensity (Figure 4.9), multicore scalability issues tend to arise from complex interactions of a hierarchical memory structure contested by many cores. This observation leads to the following fundamental difficulties when trying to trace a memory performance issue to its cause in software.

- Memory effects are highly context sensitive. They depend on the state of all the caches and DRAM banks in the system. The performance of a given function

that always performs the same amount of computation will tend to have a much higher degree of variability, and depend much more on its calling context, since performance will depend on the data the calling functions have brought on chip and the state of the memory system upon entering the function.

- Memory bottlenecks tend to be bursty. This results in extreme local variations in performance that might be hidden in larger averages.

- Memory interactions are nondeterministic. Just as debugging parallel programs can be hard due to non-determinism, memory bottlenecks caused by the interactions of threads on the memory system also tend to be nondeterministic.

- Memory bottlenecks can be highly non-local. Because the latency of the memory system can be hundreds to thousands of cycles, and because memory access speed is affected by the activities of previous memory operations, an apparent memory bottleneck often is caused by earlier functions. Additionally, the natural time skew between the activities of different cores and chips tend to spread the range of memory effects far beyond the time of complete cache turn over, the time it takes for all data in the on-chip memory hierarchy to be replaced with new data.

- Disturbing memory behavior in the attempt to measure it is much easier in the multicore regime, due to the difficulty in bracketing memory effects. The influence of a performance measurement may be felt much later, while the counter events recorded during the actual timing calls may have been caused by code in an earlier timing interval or may even have been influenced by a different core running different code. Any sophisticated timing library will need to make multiple round trips to main memory to update counter totals, resulting in time dilations of thousands to millions cycles, which along with

such effects as prefetching and cache interference can create disturbances in the entire memory system that last for tens of billions of cycles. More details on this appear in the following subsection. For this reason, we have found that developing or utilizing extremely light weight timing libraries is required to see an optimization's effects on the running code and not simply the optimization's effects on the timing code.

As a result of these fundamental issues, the notion of a given function having "average performance characteristics" is less certain, and the performance effects of the measurements themselves are much more difficult to isolate and remove. The next subsection discusses these issues in more detail and suggests techniques to overcome them.

Effects of Core Skew

An important performance effect in the multicore regime is what we call "core skew" effects. Because each core might be in a slightly different phase of execution, different functions may be running on different cores at the same time. This effect can be due to natural drift between synchronizations, such as from NUMA effects or non-deterministic memory contention, or due to an intense memory event having a somewhat serializing effect, pushing the cores further out of phase. Finally, this effect can be caused by a single core performing unique duties, such as printing results or doing performance monitoring, that kicks that core further out of phase relative to the others. The effect is that an intense event gets "smeared out" over time, because it remains in operation from the time the first core starts the section until the last core finishes it. At a phase change boundary, light computational functions may slightly overlap with memory intensive functions, slowing their performance.

Figure 4.10 shows all these effects in HOMME. The figure demonstrates the average execution time of a single function, PreqRobert, for each major computa-

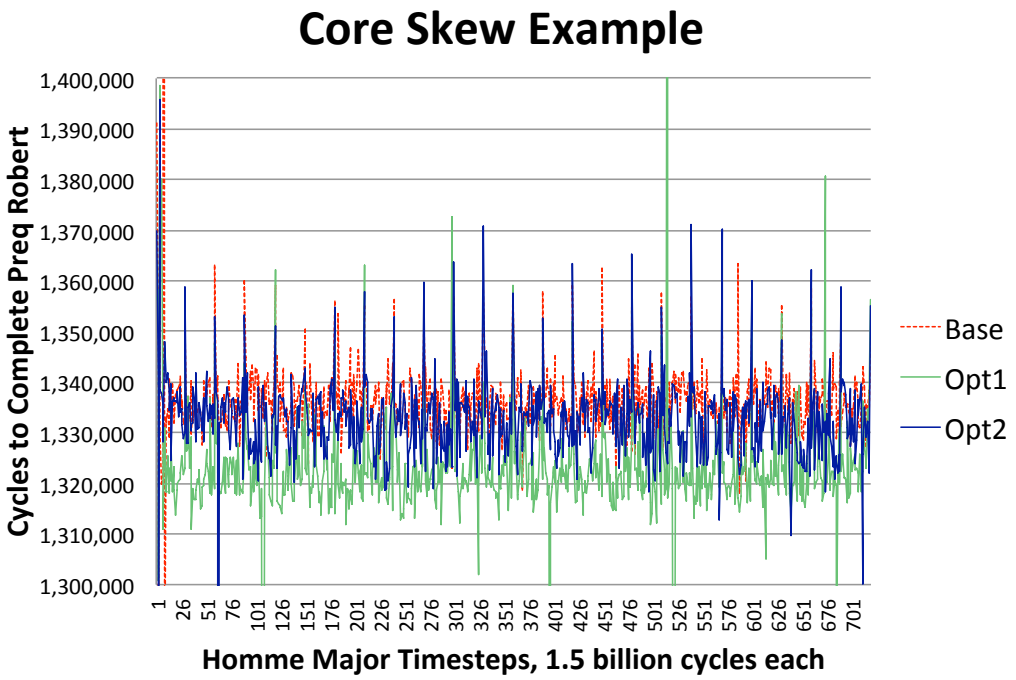


Figure 4.10: Core skew causes significant jitter and prolongs disturbances for tens of billions of cycles, making temporal context critical in measurement.

tional timestep, across a trillion cycle window. These are not short term performance fluctuations, but averages of hundreds of executions over more than a billion cycles. In particular, the extreme oscillations caused by program initialization (not shown on graph) still cause performance oscillations of 35% (clipped in graph) tens of billions of cycles later. The latent perturbations are large enough to throw off total performance averages dramatically. Note that these perturbations last many times longer than the complete turnover of all data in the memory system, which occurs many times per major timestep, and even longer than a single major timestep, during which all cores resynchronize execution through a local barrier.

In addition, because the continuous performance oscillation (on the order of a few billion cycles, or 3 timesteps on Ranger and 8 timesteps on Longhorn) is larger than the difference in average performance, average performance values do not correctly convey which optimization is better. While this graph demonstrates background oscillations of just a few percent, we have observed periodic oscillations up to 20%.

Many people already know to fast forward over the initialization and cache warm-up time. But in the multicore regime, one must fast forward *tens of billions of cycles* into regular code execution for the skew disturbances to die down. It is also critical to minimize clock skew by having all cores do as similar operations as possible.

Performance oscillations and extreme perturbation delays combine and make it critical for performance measurements to preserve temporal context. The variation spikes on the left side of Figure 4.10 are so high that the average performance of many key functions depends on exactly when the timing starts. This is why for HOMME, temporal context was much more important than calling context. Another classic example where temporal context is critical is when touching a page for the first time causes TLB misses with page initializations taking 25,000 cycles each. If the system

in use does not support huge pages, initialization can consume a large fraction of execution time.

Because conventional instruction logging is completely impractical for functions that recur tens of millions of times a second or more, we developed a hierarchical scheme in which performance metrics were averaged over sub millisecond periods, and then these period averages were used to accumulate statistical information about variation with time.

Minimizing measurement disturbance

Many measurement tools are very heavyweight, but rely on the concept of sampling versus explicit measurements. When called, such performance libraries will interrupt the program, crawl the user stack to determine the point of execution and the call graph, read out all of the performance counters, and update all the local statistics. In so doing, the entire on-chip memory system tends to get flushed and must be reloaded when the program resumes. The performance disturbances caused by the program demand paging all of its data upon resuming impacts performance for a significant amount of time. The idea is to mitigate huge perturbations to the code by only doing the measurement once in a while,no more than at most every 50,000 cycles, and preferably only once every several million cycles. Even with such long intervals, it is still not uncommon for many tools to dilate execution time several, if not dozens of times over [1], and due to the nature of memory disturbances, it is impossible to "bound" this disturbance and remove it from the performance analysis. Worse yet, the very notion of sampling to characterize an application implies statistical convergence, and in this chapter we show that the sparse sampling necessitated by heavy handed measurement software fails to uncover significant performance bottlenecks as a result. Finally, these kinds of memory heavy approaches to program analysis can lead to serious disturbance issues when

function size	% exec time
2,000 cycles or less	20%
2,000 to 10,000 cycles	10%
10K to 200K	15%
200K to 1 million cycles	15%
1 million to 10 million cycles	0%
10 million or more cycles	35%

Table 4.1: Duration of important HOMME functions

averaging many samples over many functions by completely changing the behavior of the memory system. This paper has already shown that memory disturbances can last for billions of cycles. The result is that perceived bottlenecks and optimization solutions are actually optimizing the measurement code. In the rest of this section, we show how the most popular open source performance tools overlooked important performance characteristics in HOMME.

On modern CPUs, cycle latencies are much larger. A single assembly instruction to read one performance counter takes 9 cycles. Reading 4 counters at once takes 30 cycles and is done twice per interval. A user function call can take 40 cycles. A running total must be updated, and in typical HPC functions, will be flushed from the cache between calls. Even a single low level user call to PAPI_READ to get a performance counter value takes 400 cycles, and a system call can take 5,000 cycles. Yet, even these calls are trivial compared to more heavyweight timing code that may take tens of millions of cycles and completely disrupt the memory system.

In contrast to this trend, Table 4.1 shows the distribution of lifespans of the most important HOMME functions. Roughly half of HOMME's run time (45%) is spent in very small functions, with 20% of execution time spent in functions only 2,000 cycles long, less than a single microsecond. Note that only the largest category in the table includes functions that run for 1 millisecond or longer. Additionally, short lived functions are called with extremely high frequency, with some exceeding

tens of millions of calls per second. And the effect of core skew means that these functions exhibit a large variation in performance characteristics. While we found conventional measurement techniques would work for relatively large functions taking several milliseconds or longer, about half the important HOMME functions require special measurement care to avoid arbitrarily wrong measurements. This is even more essential when evaluating potential code optimizations.

The important message is that correct assessment of bottlenecks requires utilizing multiple performance analysis tools to get qualitatively correct answers. Higher level measurement tools, which are simple to use but extremely heavy weight, are convenient for initial classification and for analysis of long running functions of tens of milliseconds or more. But once small, important functions are discovered, proper analysis requires a more delicate and targeted measurement approach such as custom PAPI or PerfCtr calls, or using lightweight or targeted tools like gprof and TAU. We provide a brief summary of our tool experiences below.

We found gprof [30] was the easiest tool to use and had the lowest overhead of any available tool aside from our custom PerfCtr code. gprof uses statistical sampling, but was accurate for all but the three smallest functions in HOMME. Tau had the lowest overhead of any explicit sampling tool, and could handle any kind of sampling context [74]. HPCToolkit had high overhead, but was the only tool that found code hot spots in arbitrary loop nests [1]. In fact HPCToolkit was responsible for isolating three of the top 11 most important code regions. PerfExpert [9] is a new tool designed to be as easy to use as gprof, and it worked reasonably well on this task. However, for our most difficult measurement tasks, we resorted to code instrumentation to measure performance counters with low enough disturbance to see meaningful results for the actual code and optimization effects, instead of just measuring the performance measurement code. We found we sometimes needed this approach even on medium sized functions taking up to a million cycles.

4.4.4 Multicore Analysis Summary

Based on the general challenges highlighted in the previous sections, we have distilled out the following recipe that provides a systematic approach to optimizing performance on multicore systems.

- Begin the analysis in a traditional way, i.e., by determining a representative run configuration and using a tool like gprof to identify the most important functions by total execution time. Running this test at minimum and maximum core density will determine the functions with the poorest intracore scalability.

- Compare the IPC, FPC and LPC values for these functions at maximum core density to good values for the underlying system to determine the optimization headroom. For functions with poor performance, gather multicore performance counter information such as L3 and DRAM miss rates for minimum and maximum core densities.

- If the L3 miss rates increase with core density, focus on optimizations that minimize cache footprints or temporary variables, or serialize data accesses.

- If there are functions that, at minimum core density, greatly exceed their share of off-chip bandwidth, but capacity is not an issue, focus on loop interchanges that may allow greater data reuse, or replace stored data with redundant computation where possible.

- If there is an increase in DRAM bank miss rates with core density, focus on rearranging loops to access only a single array at a time.

- For medium and large functions that take multiple milliseconds or longer to execute per invocation, any convenient measurement tool, such as TAU, HPCToolkit or PAPI will suffice. Shorter functions require minimal overhead tim-

ing approaches, such as a selective TAU run or PerCtr code. Averages should be checked for variations at different points of the execution.

4.5 Multicore Optimizations

The goal of performance measurement and analysis is usually optimization. We consider two categories of optimizations: algorithmic optimization, in which the algorithms are changed, and source-code optimizations, which are typically local to a loop nest or function and leave the algorithm unchanged. Source-code optimizations often require little knowledge of the algorithm and are simpler to implement. Some source-code optimizations can be automatically applied by the compiler.

The previous section identifies and establishes procedures for measuring and analyzing three multicore-specific performance bottlenecks: L3 capacity, off-chip bandwidth, and DRAM page misses. Algorithmic optimizations for alleviating these bottlenecks include making an algorithm more computationally heavy or helping reduce off-chip bandwidth and bus contention. But algorithmic optimizations often require more detailed program comprehension, spread across large code sections, and may not be portable across applications. Therefore, we concentrate on source-code optimizations.

The most effective local multicore optimization we found is *microfission*, which is a specialization of two well-known compiler optimizations, loop fission and loop fusion [5]. This optimization first splits (fissions) complex loops that reference multiple arrays into simple loops such that in each loop nest, (1) no more than two independent arrays are accessed, and (2) no more than one array is loaded from memory memory. For most linear computation, a complex calculation can be broken down into individual steps of the form $C[i] = f(C[i], X[i])$, where $C[i]$ is cached and $X[i]$ is streamed. Afterwards, the optimization combines (fuses) all loop bodies that operate on the same two arrays. For example, Figure 4.11 shows a key

```
do k=1,nlev
 do j=1,nv
  do i=1,nv
   T(i,j,k,n0) = T(i,j,k,n0) + smooth*(T(i,j,k,nm1) &
                 - 2.0D0*T(i,j,k,n0) + T(i,j,k,np1))
   v(i,j,1,k,n0) = v(i,j,1,k,n0) + smooth*(v(i,j,1,k,nm1) &
                 - 2.0D0*v(i,j,1,k,n0) + v(i,j,1,k,np1))
   v(i,j,2,k,n0) = v(i,j,2,k,n0) + smooth*(v(i,j,2,k,nm1) &
                 - 2.0D0*v(i,j,2,k,n0) + v(i,j,2,k,np1))
   div(i,j,k,n0)  = div(i,j,k,n0) + smooth*(div(i,j,k,nm1) &
                  - 2.0D0*div(i,j,k,n0) + div(i,j,k,np1))
  end do
 end do
end do
```

Figure 4.11: Loops in HOMME typically iterate over many different arrays at the same time (code shows loop from PreqRobert update).

loop in HOMME that accesses four different arrays and 24 different array sequences in a single loop. Figure 4.12 shows how the first line of the loop body is broken up so that each time only one array is brought into the private cache and at most one array is kept in the private cache.

This optimizations offers two critical benefits to code executing on a multicore chip or multichip node. First, it reduces the loop-level working set to two arrays. This may significantly reduce L3 cache misses. Second, it reduces the total number of independent locations being requested from the memory system across all the cores in a node. The result is a reduction in the number of DRAM page misses, while empowering the memory controllers to batch requests more intelligently. Finally, compilers are good at optimizing small loop bodies and thus end up producing better code once microfission has been applied.

A practical complication is that modern compilers are "smart" enough to fuse small loops and undo the optimization. To save the microfission optimization from such compiler transformations, we encapsulate each micro-loop in a separate function. While this process introduces a substantial CPU overhead (which could be avoided by better control of compiler optimization), it still improves performance. This result clearly suggests defining a compiler switch or pragma to disable loop

```
do k=1,nlev
 do j=1,nv   ! Load T(i,j,k,n0) into cache
  do i=1,nv  ! May need to block across all loops in T
    T(i,j,k,n0) = (1.0 - 2.0*smooth) * T(i,j,k,n0)
  end do
 end do
end do

do k=1,nlev
 do j=1,nv
  do i=1,nv
   T(i,j,k,n0) = T(i,j,k,n0) + smooth * T(i,j,k,nm1)
  end do
 end do
end do

do k=1,nlev
 do j=1,nv
  do i=1,nv
   T(i,j,k,n0) = T(i,j,k,n0) + smooth * T(i,j,k,np1)
  end do
 end do
end do
```

Figure 4.12: Applying microfission to the first line of the loop body in Figure 4.11. At any one time, one array stays in the private cache while a second array is streamed in.

fusion.

One important assumption of microfission is that two arrays fit completely in one core's share of the on-chip caches. For the HOMME data sets used, this is the case, as each array for a single blocked element takes up just 48KB. If the arrays were too large to fit in the caches, blocking would be necessary.

The performance effects of microfission are illustrated in Figure 4.13 as measured on Ranger. The second set of bars (L2 MR) show that the L2 miss rate is essentially unaffected, increasing slightly from 7.4% to 7.9%, despite the increased private cache bandwidth incurred by the optimization. This confirms that private cache bandwidth is ample to support the lowest arithmetic intensity HPC codes, and that microfission is a multicore optimization having minimal effect on individual cores. The next bars to the right (L3 MR) illustrate the reduction in contention for shared on-chip resources. L3 miss rates, typically above 50% for HPC applica-

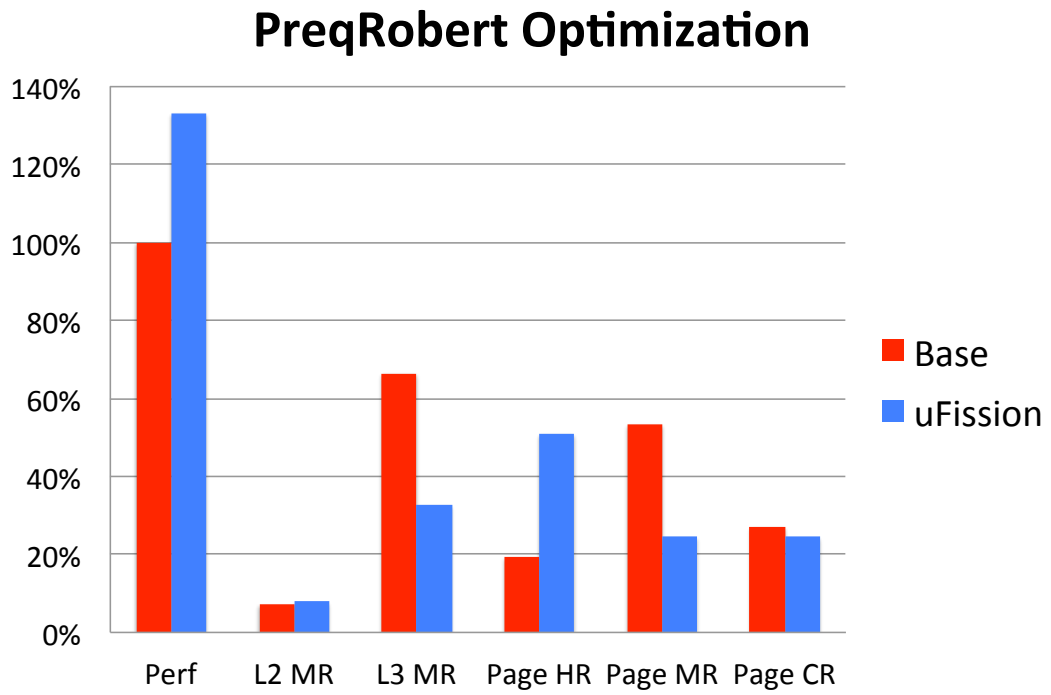


Figure 4.13: Effect of performing the microfission optimization: L3 miss rate and off-chip BW cut in half, DRAM page hits more than doubled, and performance increased by 35%

tions, are cut in half from 66% to 33% due to the local drop in working set size. Off-chip bandwidth needs are reduced accordingly. The DRAM page hit rate (bars labeled Page HR), previously below 20%, is 2.5 times as high, breaking the 50% mark, while the page miss rate (Page MR) is down from 53% to 25%. DRAM conflict miss rates (Page CR), representing the worst type of DRAM contention, are down by a more modest 11% from an already low rate of 27%. As a result of reducing resource contention, actual performance (the first set of bars) increased by 33% on Ranger and by 35% on Longhorn.

Microfission also improves intrachip/intranode scaling. Comparing the optimized code at 4 cores per chip to the base code at 1 core per chip, we see a 33% increase in scalability – on Ranger/Barcelona, the optimized code reaches 78% efficiency at 4 cores per chip and actually manages to benefit from 3 cores per chip. On Longhorn/Nehalem, the optimized version has 10% better intrachip scaling properties over the unoptimized version. However, comparing the optimized version at 4 cores per chip over the unoptimized base at 1 core per chip, the efficiency increases to 58%, a 42% improvement in intrachip scalability that is able to benefit from all 4 cores.

The computational pattern that enables microfission, i.e., multiple terms in an equation discretized on the same grid, is common across regular HPC applications so that microfission should enhance the performance and scalability of many such applications. The detailed measurement and analysis of the performance of HOMME on multicore chips and the success of microfission suggests some guidelines for coding regular applications for multicore chips: (i) employ structures of arrays rather than arrays of structures, (ii) group all computations on a logical data structure in the same loop as much as possible, (iii) minimize the number of temporary arrays that are declared and used, and (iv) adjust array/loop block sizes to fit in the caches that are characteristic of multicore chips.

4.6 Summary

This paper details an experimental study of intrachip/intranode scalability and the factors that determine this scalability for HPC applications running on systems with multicore chips and multi-chip nodes. Our study focuses on a deep analysis of HOMME, an at-scale application used for production climate modeling.

In line with our thesis, we show that throughput performance is different than single threaded performance. Our results demonstrate that effective measurement, analysis, and optimization of memory performance bottlenecks intrinsic to multicore nodes require a different and more complex approach than memory bottleneck detection and alleviation in unicore nodes. We show that multicore systems exhibit qualitative and quantitative differences in performance bottlenecks and metrics, in experimental and measurement issues, and in optimization strategies.

For example, we show that accurate memory performance measurements in multicore environments must account for the delayed effects of memory references and the nondeterministic interactions among the cores on a chip and/or node, which are immaterial or absent in unicore nodes. We further show the temporal context of the measurements to be critical, and obtaining sufficient measurement accuracy may require using multiple tools. We demonstrate a range of measurement techniques to overcome these complications, including the creation of a light weight library for reading hardware performance counters. We also describe a structured process for effectively measuring the performance metrics critical to multicore chip and multichip node performance, including methods for interpreting these metrics to obtain an accurate definition of the causes of multicore-related memory performance bottlenecks.

Using this process, we have confirmed what the mathematical model indicated — the core performance bottleneck is the shared memory system. Specifically, significant performance bottlenecks exist in all three key shared memory resources

that would be anticipated from Chapter 3: shared L3 cache capacity, shared off-chip bandwidth, and DRAM page conflicts. Scaling tests showed that Homme was operating on the right side of the cache regime in the valley graph — in the steep drop off — and so techniques which can reduce the cache footprint of the application even by a small amount should lead to large throughput benefits and increased scalability. Driven by these bottlenecks, we developed a source-code loop-level optimization called microfission that, when applied to HOMME, reduces the L3 cache miss rate by almost 50%, more than doubles the DRAM page hit rate, reduces compiler overhead instructions by a third, and increases intrachip scalability by up to 42% and absolute performance by up to 35%. We anticipate that microfission will be applicable to a wide range of multicore applications and that our insights into multicore bottlenecks will inspire additional optimizations specifically aimed at multicore execution.

The contributions of this study included the performance characterization of a major HPC Challenge supercomputing application in the native environment of the Ranger Supercomputer, the identification performance bottlenecks in the shared memory system of the node, the realization that core-shared performance counter metrics are of key importance for multicore application throughput. And that due to the incredible lag times on multicore disturbances, multicore performance analysis requires studying cycle spans an order of magnitude longer than traditional unicore applications. We demonstrated the pitfalls of traditional, heavyweight sampling systems, and introduced a custom, ultra light weight library for reading performance counters, Finally, we demonstrate that localized multicore optimization techniques exist and introduce a new compiler technique called "Loop Fission" that lead to several follow on studies at TACC[19, 9, 20].

This coarse grained study validated what we learned from the mathematical model of throughput performance. First, that the most critical performance bottle-

neck would be in the shared memory system. The optimizations we made actually hurt single threaded performance, yet were a net performance win. And second, in line with the performance spiral, even a slight gain in efficiency delivered significant results in throughput performance. While the chaotic contention between threads in a complex multicore system would seem utterly beyond the ability of a programmer or compiler to control, just slightly increasing caching efficiency made a big difference. In this case, Loop Fission over an incredibly small time scale reduces the working set of each thread. This reduces shared cache misses, improves bandwidth efficiency, and reduces contention throughout the system. We will now see how the guiding principles of throughput performance may apply to fine-grained parallel systems.

Chapter 5

Fine-Grained Throughput Architectures

We will now attempt to apply our knowledge of throughput performance to fine-grained throughput architectures. We expect the tradeoffs to be far more severe than in the coarse-grained case, since the thread to on-chip memory ratio is far higher than with coarse grained throughput architectures. In Chapter 3, we saw that there are many ways to improve the efficiency of a cache and reduce latency. In the type of fine-grained system we now study, called a Replay Architecture, there is a unique opportunity for latency reduction. As we will explore in Chapter 6, in this kind of system, a memory instruction must continually replay form the beginning until all of its SIMD lanes complete. Anything that can help the lanes complete faster has the effect of dramatically reducing single thread latency.

In our study of coarse-grained throughput architectures in Chapter 4, we took a fixed hardware architecture and looked at how we could improve the software stack. In this chapter we take an existing software stack and look at how we can improve the hardware architecture to increase parallel efficiency and throughput. We will apply what we learned in the multicore study — that it is absolutely critical that our

metrology is sound, that we are adequately modeling the most crucial architectural elements of throughput performance, and that we are doing so on a time scale sufficient to stress the on-chip memory system.

This chapter describes our initial methodology and insights into a type of fine-grained architecture known as a Replay Architecture. This architecture is loosely based on NVIDIA GPUs, although not intended to represent any specific product. We begin by describing our custom GPU simulator along with our rationale for creating one. Then we describe our target fine-grained hardware architecture along with a basic description of a Replay Architecture. Finally, we describe the Rodinia and Parboil parallel benchmark suites we use in our evaluation, providing characteristics of the memory access patterns of these benchmarks that are critical to guiding architectural enhancements in Chapter 6.

5.1 A Custom Simulator

We created a custom simulator to address some fundamental issues with existing throughput simulators. The following were are primary design goals.

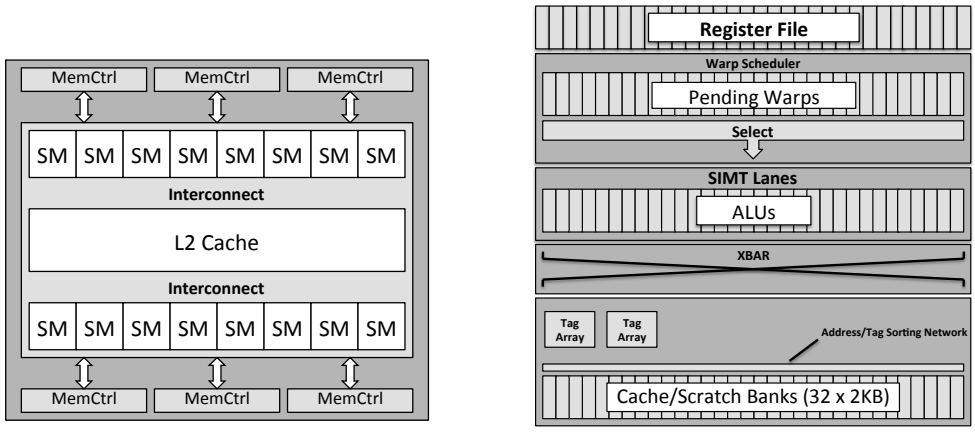
- Model a plausible on-chip memory system for a throughput architecture. No existing simulators had timing models for the primary on-chip memory systems of throughput architectures, many modeling 1-cycle latency for access. This trivializes the need for TLP, dramatically skewing results in a wide range of research such as thread scheduling.

- Speed. Simulators that are too slow make it difficult to analyze throughput memory systems with scales that could stress the memory system. As discussed in Chapter 3, throughput architectures require much longer run times to get meaningful performance data.

- Robustness. How easy is the simulator to debug? How do we know we are getting the correct result?
- Software issues. Many simulators at the time used compiler intermediate code as a proxy for true assembly language. Compounding this issue were poor optimizations at this stage in the tool chain that substantially misrepresented metrics like memory intensity and TLP.
- Baseline performance. It's easy to show an architectural speedup over a naive baseline. We wanted to make sure we had a very aggressive baseline architecture so that any speedup results would be significant.

Our simulator is a custom GPU architecture simulator that models long latency replay mechanisms, sort networks, crossbars, and a GPU-like primary memory system. Our simulator is a fully execution-driven, highly detailed, cycle-accurate simulation of the entire GPU core and secondary memory system. We fully model the execution pipelines, thread selection and retirement, multiple pipelines in the primary memory system, tag and bank access, re-order buffers, sort networks and crossbars, wire traversals, and contention at all levels. Our performance numbers are the total cycles taken by each application as simulated; all benchmarks are simulated to completion.

For efficiency, we leverage a modified version of Ocelot [43, 23] for the interpretation and functional execution of instructions in CUDA programs. Ocelot then feeds a cycle-accurate timing simulation that is fast enough to run the largest possible datasets on all benchmarks for billions of cycles, realistically stressing the memory system. We ran benchmarks on a single SM and allocated them a single, direct mapped, 48KB slice of the 768KB L2 cache. This simplification results in somewhat conservative L2 cache performance, since SMs cannot benefit from shared data. However, unlike the findings in [39], we found that global data sharing across



(a) GPU chip (b) Streaming multiprocessor

Figure 5.1: Baseline GPU architecture.

tasks (CTAs) is minimal in the applications we evaluated.

Power model: Modeling power requirements is challenging, as the structures examined in this paper are not modeled in conventional simulators [50], and the area and power required depend highly on the exact VLSI implementation. For replay costs, we estimated core pipeline power and redundant register reads as a fixed fraction of SM power in the manner of [28]. Power in the primary memory pipeline is dominated by the cross-bar switch, whose power is best approximated by a wire-transmission model. We modeled the area and power of the cross-bar similar to CACTI [75], using a standard cell library for geometry and computing active power based on actual wire traversals during simulation. We chose the conservative VLSI layout of a monolithic grid connecting data lanes to SRAM banks, and modeled active and static power using ITRS Roadmap wire models, taking the approach of [47]. The power needed for register accesses, tag accesses, SRAM banks, and L2 caches, as well as the size and power for crossbar logic came from standard cell data reported in [31].

5.2 Replay Architectures

Our baseline architecture is loosely modeled on NVIDIA's Fermi architecture, shown in Figure 5.1. This is a generic design similar to others in the literature [6, 62, 84], and is not intended to correspond directly to any existing product. The GPU consists of 16 *streaming multiprocessors* (SMs), each containing 64 SIMT (single-instruction, multiple thread) lanes that each execute up to one thread instruction per cycle. Each SM dynamically manages up to 8 different CTAs (Cooperative Thread Arrays) simultaneously, populating up to 48 warp slots with up to 1,536 total active SIMT lanes. Up to two 32-lane warp instructions per cycle issue in order and complete out of order. Each warp controller contains a small instruction buffer and scoreboard to prepare instructions for issue, and a reorder buffer for in-order retirement.

Each SM contains a 128KB SRAM array for storing register values and a 64KB SRAM scratchpad, half of which is used as a 32KB L1 cache. The baseline SRAM array is divided into 32 fully independent 32-bit wide banks and supports full scatter/gather via a bank sort network and a 32×32 32-bit crossbar. Global tag arrays determine if an L1 cache line is present. Each L1 cache line is 128-bytes, spanning 32 banks. Our L1 cache is direct mapped. While slightly pessimistic, GPUs have limited associativity to conserve power, and AMI provides an alternate approach to reducing set conflicts. L1 misses coalesce in MSHRs and are sent to a 768KB global L2 cache. L2 misses go directly to DRAM. As all our benchmark kernels fit in a few KB, we ignore instruction cache misses.

Active lanes in the warp may not all successfully complete their memory transaction due to (1) a conflict in the tag array (too many cache lines referenced), (2) memory divergence (not all cache lines present in L1), or (3) bank conflicts. Incomplete memory instructions move to a replay buffer which contends with new instructions for issue slots. To preserve memory consistency, other in-flight memory

instructions from the same warp are squashed and reissued. Our model is more aggressive than current GPUs, which avoid a replay queue by squashing and reissuing all instructions from the warp beginning with the conflicted memory instruction [56, 57]. Replays reuse major SM hardware and preserve memory consistency but also cost energy and reduce throughput performance [78].

To demonstrate a meaningful improvement over a state-of-the-art architecture, we chose an aggressive base case with dual global tag arrays that virtually eliminates tag conflicts while increasing average base performance by 10%.

5.3 Fine Grained Benchmark Suites

After studying throughput applications in the context of conventional CMPs, we decided to look at applications designed for fine-grained throughput architectures. As before, we decided that it would be useful to examine the characteristics of more regular, legacy throughput code designed for earlier GPU architectures before moving on to less regular, more general purpose throughput code. These applications historically ran very efficiently on GPUs and were designed for earlier, simpler architectures without on-chip caches or scatter gather ability. The common assumption about such code is that it would not benefit from low latency throughput architectures because there would likely be high arithmetic intensity, little data locality (aside from scratchpad use), and massive amounts of thread level parallelism to mask latencies to main memory. The results were mixed, but surprising. A lot of the applications had significant reuse, at least within a task. Yet access was less regular than might be assumed, with cooperative threads being more independent than previously assumed. Some had relatively low arithmetic intensity, and most had a surprising lack of available parallelism preventing them from scaling up to larger throughput chips without increasing parallel efficiency.

To cover a wide range of applications in a short time, we chose to target

GPGPU benchmark suites, but to run them with the largest possible input datasets, to stress the on-chip memory system in a realistic way. Parboil [65] and Rodinia [12] are the two most widely used GPGPU benchmarks in academia, and they differ markedly from CMP style throughput benchmarks like PARSEC in their use of much finer grained parallelism and independent (vs coordinating) tasks. While other publications have examined coarser aspects of their behavior [13], we have studied their data access patterns in detail.

The aim of this characterization study is to determine the degree to which parallel efficiency in these benchmarks could potentially be improved through the on-chip memory system. Answering this question requires examining the performance characteristics identified in Chapter 3 as the most critical to throughput performance:

- What is the dynamic instruction mix of the applications, i.e., what percentage of instructions actually access memory in some way? Arithmetic intensity and the overall dynamic instruction mix has a first order effect on the sensitivity of throughput performance to instruction latency, and it can be altered by the algorithm, the compiler, and the ISA.

- How is this data from memory used? Is there enough reuse to offset compulsory (cold) misses? Is there enough concentration of locality in the working set to effectively cache on chip? Is there enough spatial locality (contiguity) in the working set to be efficiently exploited by cache lines? How common is data sharing across different tasks?

- What is the data sharing behavior between individual MIMD threads within each task? This impacts the ability to reduce local working set size through different thread scheduling algorithms.

5.3.1 Rodinia and Parboil Benchmarks

This section provides a brief overview of the benchmark applications, highlighting their diversity. The following sections highlight some key performance characteristics relevant to our study.

The Parboil benchmarks were created by the IMPACT Research Group at the University of Illinois. They are kernels adapted from classical CMP style coarse parallelism, with longer tasks operating on more data. The largest Parboil benchmarks executes 350 billion scalar instructions. The Rodinia benchmarks have also been hand translated from conventional parallel code, but have been optimized using the styles of classical GPGPU programming. Rodinia is a living benchmark suite that is constantly expanding with new applications designed to have unique characteristics. Currently, we have analyzed a subset of 13 benchmarks that have already been ported to the Ocelot framework.

Both of these benchmark suites are largely representative of GPGPU past - they target applications that have been widely successful at leveraging GPUs and tend to have fairly regular computation, with no assumptions of on-chip cache. We assume they are characteristic of legacy GPGPU applications.

Parboil Benchmarks

- **sad** - Sum of absolute differences. An integer benchmark in the area of image analysis. Based on the full-pixel motion estimation algorithm found in the JM reference H.264 video encoder.
- **cp** - Distance Cutoff Coulombic Potential - Computes the short-range component of Coulombic potential at each grid point over a 3D grid containing point charges representing an explicit-water biomolecular model.
- **tpacf** - Two Point Angular Correlation Function - Used to statistically analyze the spatial distribution of observed astronomical bodies. The algorithm

computes a distance between all pairs of input, and generates a histogram summary of the observed distances.

- **mri-q** - Magnetic Resonance Imaging - Q - Computes a matrix Q, representing the scanner configuration for calibration, used in a 3D magnetic resonance image reconstruction algorithms in non-Cartesian space.
- **mri-fhd** - Computes a regular grid of data representing an MR scan by weighted interpolation of actual acquired data points. The regular grid can then be converted into an image by an FFT.
- **rpes** - Molecular Dynamics simulation.
- **pns** - Petri Net Simulation - A graph traversal algorithm.

Rodinia Benchmarks

- **backprop** - Back Propagation - a machine learning algorithm used to train the weights of connecting nodes on a layered neural network.
- **needle** - Needleman-Wunch - a global optimization method for DNA sequence alignment.
- **hotspot** - Thermal simulation of a processor based on an architectural floor plan.
- **srad** - Speckle Reducing Anisotropic Diffusion - a diffusion algorithm based on partial differential equations used for removing the speckles in an image without sacricing important image features. Widely used in ultrasonic and radar imaging applications.
- **lu** - LU Decomposition from Dense Linear Algebra.

- **hwt** - Heart wall - an image tracking program that operates on an AVI video of a mouse xray and tracks the walls of the beating heart.

There were multiple input data sets for these benchmarks, and we chose the largest input data sets available, both to represent a more realistic application, and to sufficiently stress the memory system.

5.3.2 Arithmetic Intensity

The most fundamental question to investigate is the arithmetic intensity of the benchmarks, as insufficient use of the memory system limits any possible benefits we can deliver. This question goes beyond the conventional view of arithmetic intensity, since most of the applications already make use of the on-chip scratch pad, filtering out both memory intensity and memory reuse from the conventional cache hierarchy. What we found confirmed a number of hypothesis - that the tool chain would artificially inflate arithmetic intensity, and that the majority of memory references were to scratchpad accesses. However, we did find that the majority of benchmarks still had enough references to conventional memory to potentially benefit from caching - if there was sufficient data reuse. With the exception of pns, for which loads and stores are roughly equal, load instructions make up the majority of memory accesses and are more critical wrt latency sensitivity.

Arithmetic intensity can be further divided between conventional, cache memory, and the explicit use of a hardware *scratch pad*, which is a common feature of these architectures. The conventional programming model of GPUs is to use a scratchpad as an explicit L1 cache, which often limits the reuse of data in conventional caches. We find that most of the benchmarks follow this pattern, with most data accesses to the scratchpad, and a lesser amount to conventional memory. The exception are four benchmarks from Parboil - mri-q, mri-fhd, cp and pns - which make no use of the scratchpad. Figure 5.3 shows the dynamic instruction ratio

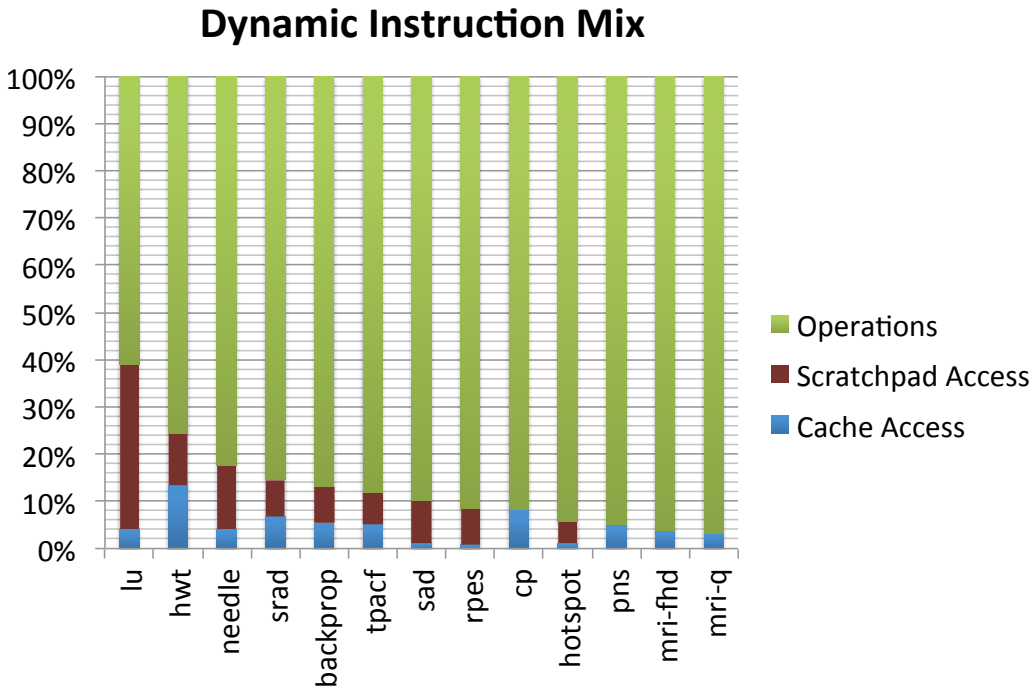


Figure 5.2: Ratio of memory instructions.

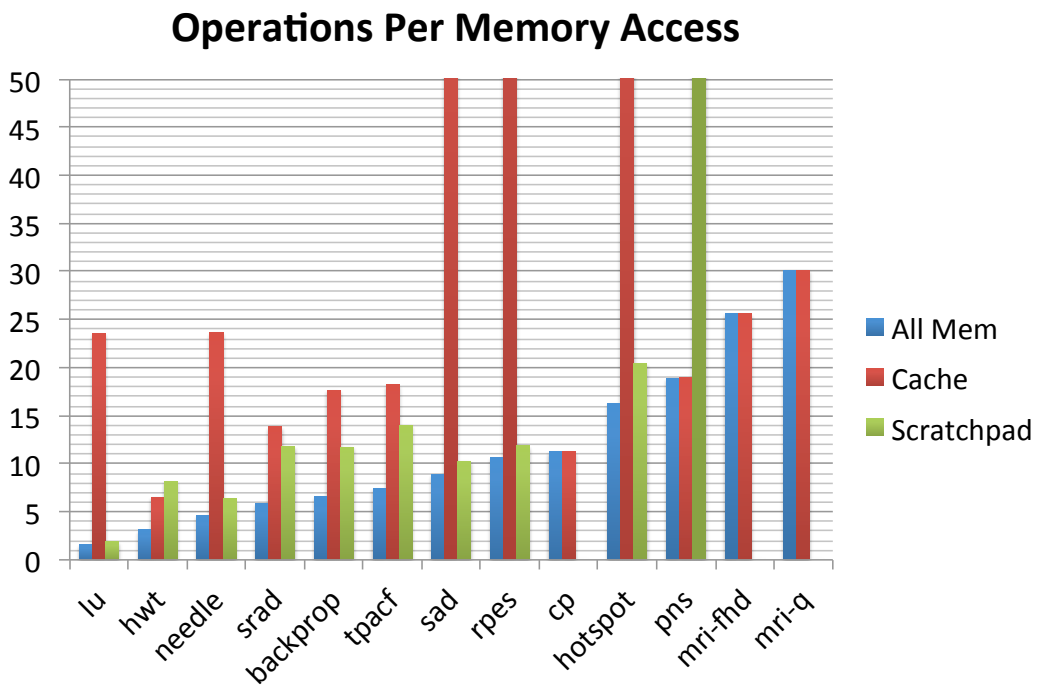


Figure 5.3: Arithmetic intensity visualized as number of operations per memory access.

between conventional memory (cache) access, scratchpad access, and operations. Figure 5.2 shows this data in terms of arithmetic intensity, that is, the ratio of operations to memory accesses of each type. The message is that conventional memory access is a far smaller ratio of instructions in these benchmarks than in the earlier CMP studies, with hwt being the only benchmark with conventional memory access above 5% of instructions. This would foreshadow a reduced role of the memory system in throughput performance, although we expect these ratios to increase once we correct for compiler and ISA effects.

5.3.3 Available Parallelism

Another very fundamental question is whether these benchmarks have ample parallelism, in which the main goal of increasing parallel efficiency is to increase throughput performance at the architectural limits of supporting parallelism, or if these benchmarks might also be in danger of parallel starvation, in which insufficient algorithmic parallelism exists to saturate the machine. The common view is that these applications, by being fine grained, have an almost unbounded amount of parallelism, which would make parallel efficiency and caching of little importance. We were surprised to discover that even the finest grained applications, running with the largest input files, actually encountered parallel starvation relatively quickly. This motivates architectural designed that increase parallel efficiency, regardless of underlying technology issues, and extends the benefits of this work beyond issues of the power and memory walls. While some applications may be rewritten to expose even more parallelism, there is a fundamental limit to available parallelism given by the amount of data accessible to a given core and the amount of operations supported by that data.

As seen from Figures 5.4 and 5.5, available parallelism is a first rate issue for the majority of the benchmarks. Current NVIDIA GPUs support 30 cores, each with

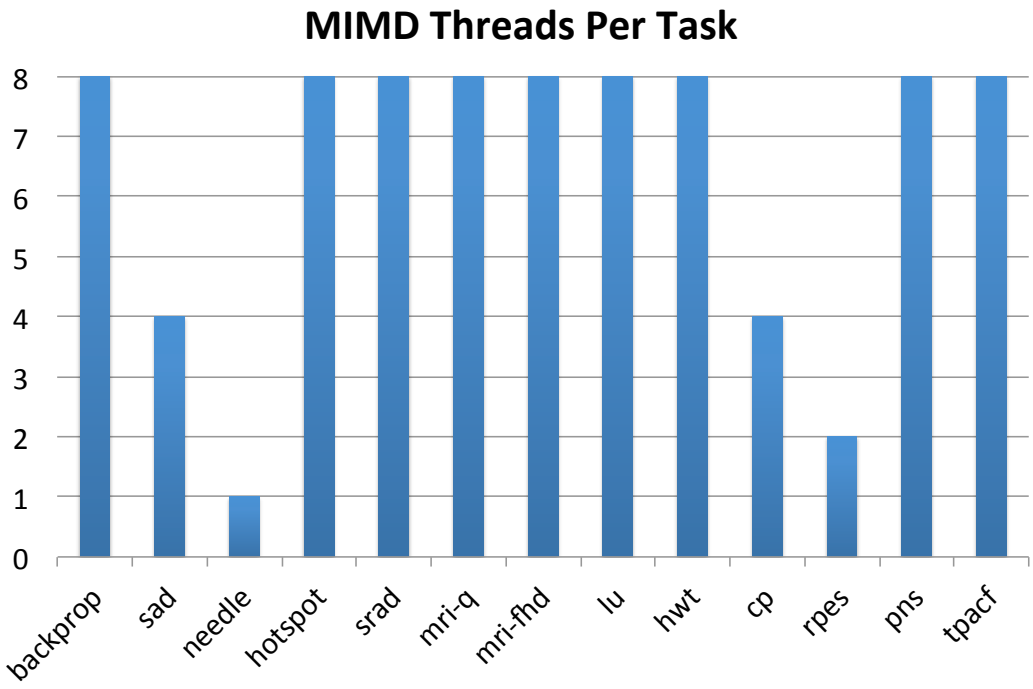


Figure 5.4: MIMD threads per task.

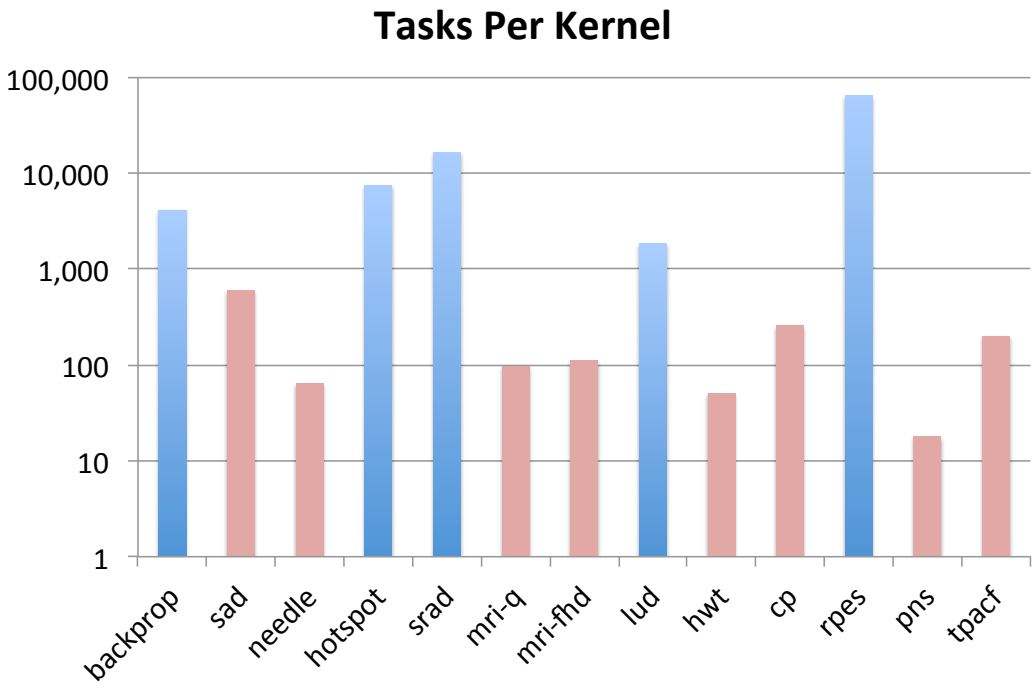


Figure 5.5: Tasks per kernel.

up to 64 MIMD threads (or "warps") to cover latency. Out of the 13 benchmarks, only five have insufficient parallelism for a current generation GPU, while only four benchmarks have sufficient parallelism to utilize Echelon. For most of the other benchmarks, parallel starvation will be an issue, which is another reason parallel efficiency is important.

5.3.4 Conventional Memory Usage

Having shown that a majority of the benchmarks actually do make sufficient use of conventional memory to potentially benefit from caching, the big question involves data reuse. Is this data merely streamed into the scratchpad, or is there enough locality left in the conventional memory system to exploit? We initially analyze the 6 benchmarks having at least 4% of their dynamic instructions accessing conventional memory. Three of the remaining benchmarks, sad, hotspot, and rpes, have around 1% of instructions accessing conventional memory, although this low memory intensity is partially an artifact of the tool chain creating redundant instructions. We found that all these benchmarks except for needle have significant per-task data reuse that we can exploit with local caching.

benchmark	cache intensity	per task WS(B)	shared WS(B)
hwt	12.73%	454,217	41,806,196
srad	6.74%	421	100,675,584
backprop	5.37%	1,600	4,456,452
pns	5.01%	8,000,009	146,172,044
lu	4.08%	1,035	16,517,120
needle	4.06%	4	16,264

Table 5.1: Working sets, conventional (cache) memory

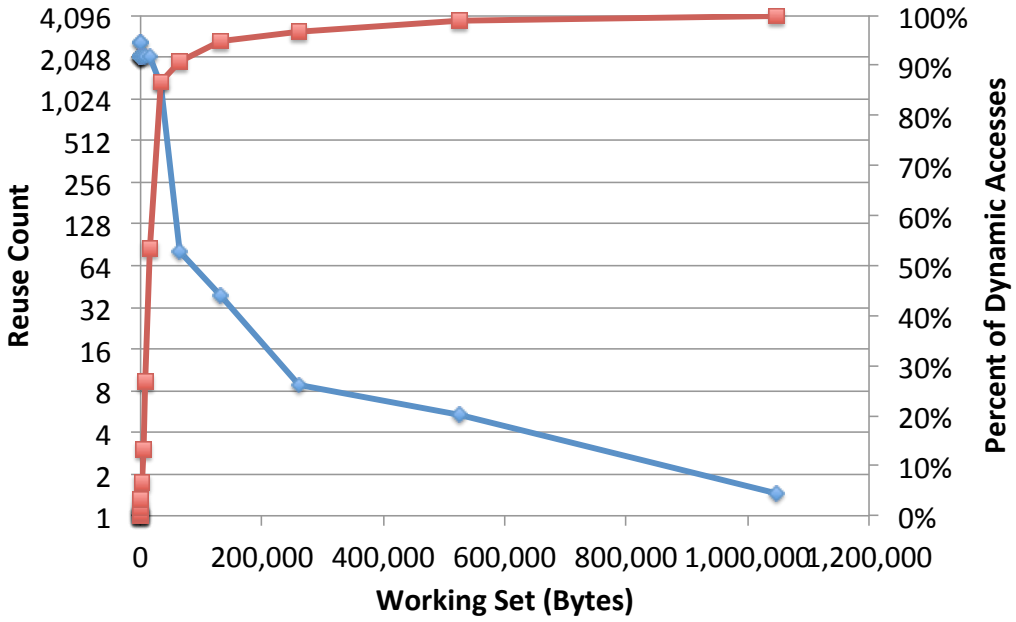


Figure 5.6: Working set concentration and reuse for hwt, the largest benchmark in Rodinia.

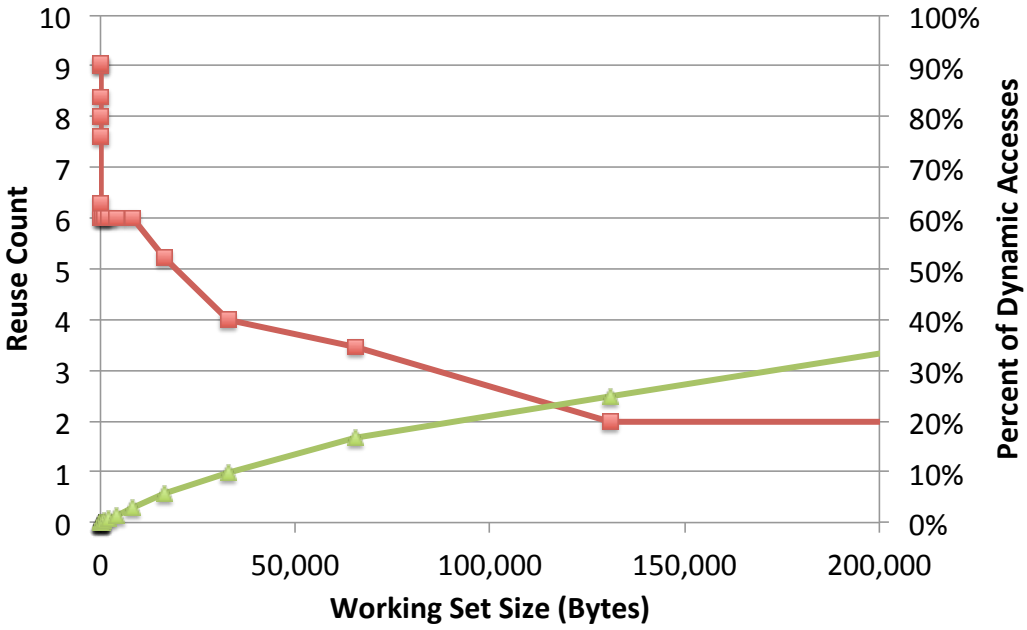


Figure 5.7: Working set concentration and reuse for pns, the largest benchmark in Parboil.

These five benchmarks show significant concentration of locality within each task, with most dynamic accesses occurring in half the working set size. As shown in Table 5.1,

only two of these benchmarks, pns and hwt, have task working set sizes greater than 2KB and so warrant a more detailed examination. Not shown here is that significant locality occurs within most of the working sets.

Figures 5.6 and 5.7 shows that even with the two largest benchmarks, hwt has enough concentration to be considered cacheable for at least a single block, with 86% of data references satisfied in 32KB, while pns has such a large working set that only 25% of dynamic references fit in 128KB. For even these benchmarks, local caches will still capture significant data reuse, although a much smaller fraction. Note that the dynamic working set is influenced by the exact ordering of the instructions and thread scheduling. So this is a somewhat approximate result, taken with the original PTX code and default Ocelot round robin scheduling. Also note that this reflects individual word accesses only, i.e., it measures temporal locality, not spatial locality. If a cache has a line size of N words, the degree of data sharing will increase by up to N times, depending on the contiguity of the data.

Dynamic working set sizes across tasks, that is, data used by more than one task, is significantly larger, and has far less locality than within a single task. Although the available cache on current throughput architectures would be insufficient to provide much benefit for any of the benchmarks except needle, by 2017 available on-chip cache would capture even the largest working sets [41]. Typically, we found most of the dynamic accesses occur in about half of the dynamic working set, or about 25% of the total working set.

5.3.5 Contiguity of Data Access

benchmark	task	inter-task
hwt	99.91%	99.87%
srad	97.00%	100.00%
backprop	86.21%	94.44%
pns	99.95%	42.62%
lu	100.00%	100.00%
needle	0.39%	3.49%

Table 5.2: Contiguity of conventional memory accesses.

The results stated thus far represent a completely ideal cache. The next question to ask is the degree to which a benchmark can use all the sequential words in a larger cache line, or its contiguity. Contiguity also represents short term spatial locality, in which multiple words can be loaded in a single line. The good news is that, within local tasks, these classic GPGPU benchmarks do tend to use most of the data in each cache line, so use of cache lines in the cache hierarchy should not significantly impact performance.

To approximate contiguity, we compare the dynamic working set sizes of the benchmarks when using 64-byte lines to those using 4-byte lines, assuming a fully associative cache with ideal replacement. If the working set size does not increase, then the benchmark utilized all the words in cache lines. Conversely, the ideal working set size divided by the 64-byte line working set size approximates the average line utilization, and hence contiguity of each benchmark. Shown in Table 5.2, we see that most of the benchmarks make extremely complete use of cache lines. The exceptions are pns at the inter-task level, which only utilizes about half the cache lines, and needle, which seems to have pathological line behavior.

5.3.6 Scratchpad Usage

Finally, what about the scratchpad usage itself? It would be easy to assume that programmers put data in the scratch pad because it is reused, and so there is little we could do to improve upon this explicit specification. However, investigating actual scratchpad usage was the most surprising of all. We found concentration of data locality within the scratchpad to be so high, that the scratch pad itself could be augmented by an extremely small, high performance cache, with important data replicated where needed to increase access efficiency. This opens up a wide range of potential architectural improvements that encompass the scratchpad into the general on-chip memory hierarchy.

As shown in Figure 5.2, the scratch pad is typically accessed more often than conventional memory, so it is worth investigating its characteristics. We found that these benchmarks typically did not use much shared memory per task - typically a few KB for the finer grained benchmarks up to 16KB for the larger ones. Yet, even within that small amount of memory, locality was so extreme that just a tiny fraction of addresses (ranging from one to 128) represented anywhere from 60% to over 90% of all dynamic accesses.

5.3.7 Memory Access Patterns

Stereotypical views of throughput applications include extremely high arithmetic intensity (low memory intensity), little to no reuse of data (streaming applications), and regular, sequential data access, which would make notions of bank conflicts irrelevant. However, we find that the majority of these benchmarks exhibit more intense and more complex memory behavior.

Figure 5.8 shows how values are reused across threads and warps for benchmarks with more than 0.01% *memory intensity*, defined as the fraction of instructions that are `load` or `store` operations. Some benchmarks have two distinct phases,

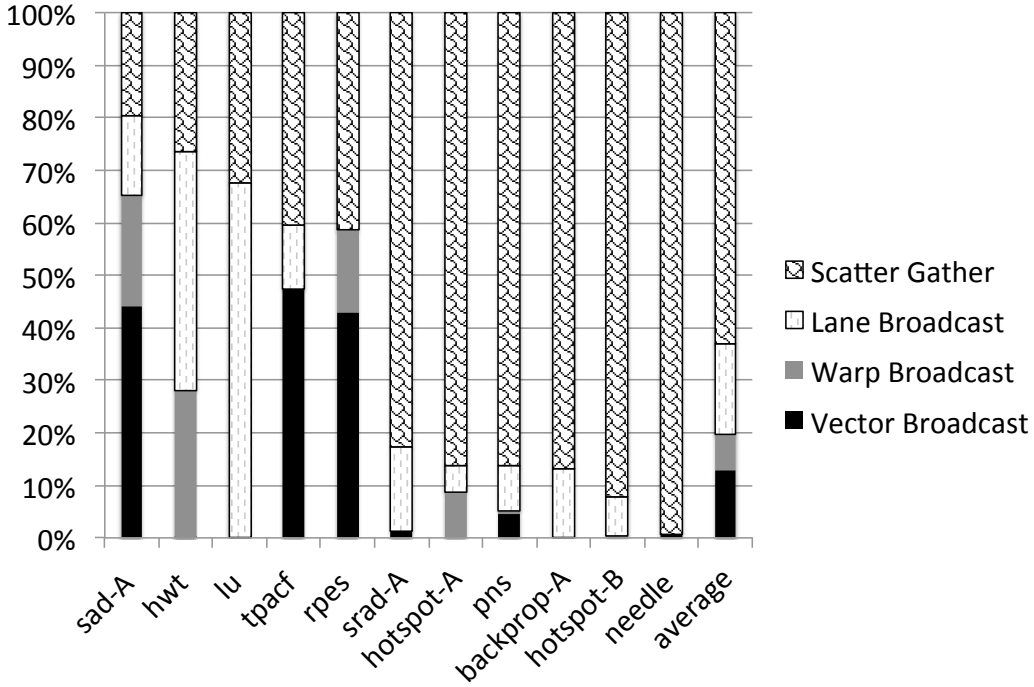


Figure 5.8: Categorization of GPU memory accesses.

labeled A and B. We found most inter-task (CTA) sharing of data was among small groups of tasks separated by large distances in space and time, requiring complex task scheduling to leverage.

In the figure, a *vector broadcast* represents a CTA-wide access to the same value that need only be fetched from the memory system once. A *warp broadcast* represents the same concept, but limited to the extent of a warp (32 threads). A *lane broadcast* represents the same concept but for a partial warp. Finally, *scatter/-gather* represents individual thread accesses to different addresses. Roughly half the benchmarks have more than 50% of all memory instructions as some form of scalar broadcast, which results from an architecture that has a vector nature, but lacks scalar registers to span vector elements. As the architecture already coalesces these accesses to eliminate unnecessary bank conflicts, we do not consider them in our bank conflict analysis. A second observation is that a significant fraction of accesses

cannot be completely coalesced (as emphasized by lane broadcasts and part of the scatter/gathers), subjecting them to both tag and bank conflicts.

5.4 Summary

This section illustrates some useful and surprising observations about the applications in classic GPGPU benchmark suites, that provide insights into future bottlenecks.

Some intuitions were confirmed. GPGPU-Applications tend to rely more heavily on the scratchpad than the general purpose memory system (L1 cache), and access patterns tend to be more regular for cache accesses and extremely random for scratchpad accesses. (But note that not all applications use the scratchpad.)

However, the results also dispel some common myths. Many GPGPU applications, similar to HPC applications, have relatively high memory intensity, and this memory intensity would be even higher were it not for an artifact of the CUDA to PTX compiler. It is also common to see a very high level of data sharing across all lanes in a kernel-instruction, although this is primarily an artifact that this parallel architecture does not contain a shared, scalar register file. In this sense, whether a memory operation represents a scalar or vector is highly correlated with the PC address. This presents an opportunity to accelerate bulk loads and stores based on the high degree of broadcasting of values across all lanes. In fact, we saw a concentration in the use of scratchpad working sets so significant that it might have been more appropriate to use indexible registers.

We did not see an abundance of available parallelism in these applications as written, making parallel starvation (insufficient TLP to cover latencies) a real issue and highlighting the importance of lowering average instruction latencies. We also saw a wide range of working set sizes, indicating that different benchmarks benefitted from different levels of parallelism. However, it is important to note that these

benchmark suites were designed for older GPUs, and that emerging, sophisticated GPGPU applications may have different characteristics and performance needs.

The most important observation is that even the largest memory intensive applications seem to have sufficient locality and reuse to benefit from caches. GPUs did not initially have traditional caches, but aded them in to aid general purpose programs. However, in recent years, the amount of cache per thread has been reduced. It may be worthwhile to increase the amount of L1 cache and scratchpad space in future systems, and to try to reduce the hit latency.

We will now apply what we have learned to augmenting the hardware of Replay Architectures to make them more efficient.

Chapter 6

Arbitrary Modulus Indexing

The goal of this dissertation is to find simple hardware optimizations that can apply to most facets of the on-chip memory system and increase throughput efficiency for both coarse and fine-grain throughput architectures. Unlike the results from Chpater 4, we discover that in a Replay Architecture, cache efficiency can be more effectively increased not by focusing on working set size and hit rate (which already shows great concentration), but rather on effective hit latency. We demonstrate that the primary performance bottleneck in this architecture is conflicts in the shared on-chip memory system which lead to instruction replays and effectively multiply the already large hit latency of the primary on-chip memory system. As described in Chapter 3, hit latency linearly reduces the throughput benefits of caching. By reducing conflicts in the primary on-chip memory system, we can greatly reduce the effective hit latency and, in line with the mathematical results, show a significant improvement in throughput performance.

In this chapter we derive a localized hardware modification that removes

*Information in this chapter was previously published in the IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2014 [22]. J. Diamond was lead author and performed all research and partial research funding, S. Keckler supervised and provided partial research funding, and Don Fussell supervised and provided help polishing the text.

performance pathologies throughout the on-chip memory system, and then evaluate the efficacy of this technique in the context of a Replay Architecture. Analysis is done using the simulator and benchmarks described in Chapter 5. In keeping with the lessons of Chapter 3, we discover unique opportunities to lower the effective latency of memory instructions even more than approaches that increase hit rates. Instead, we augment on-chip memory in a way that reduce conflicts in the memory system.

In essence, computers use binary arithmetic. As a result, nearly all physical structures in computers tend to be power-of-2 sizes. Programmers optimize for these structures, leading to power-of-2 sized references and strides. This tends to lead to extreme pathologies due to mapping conflicts. As a result, architects must resort to either unaffordably expensive CAM style technology which can lookup any position any time, or accept the complexity and mediocre performance of an ad hoc address mangling scheme. This is no longer the case. By finally generalizing the hardware and mathematical solution of Mersenne Primes to all integers, we allow simple hardware to map to non power-of two-locations, breaking the performance pathologies caused by power-of-2 sized references.

Modern high performance processors require memory systems that can provide access to data at a rate that is well matched to the processor's computation rate. Common to such systems is the organization of memory into local high speed memory banks that can be accessed in parallel. Associative look up of values is made efficient through indexing instead of associative memories. These techniques lose effectiveness when data locations are not mapped uniformly to the banks or cache locations, leading to bottlenecks that arise from excess demand on a subset of locations. Address mapping is most easily performed by indexing the banks using a $\mathrm{mod}(2^{\mathrm{N}})$ indexing scheme, but such schemes interact poorly with the memory access patterns of many computations, making resource conflicts a significant mem-

ory system bottleneck. Previous work has assumed that prime moduli are the best choices to alleviate conflicts and has concentrated on finding efficient implementations for them. In this paper, we introduce a new scheme called Arbitrary Modulus Indexing (AMI) that can be implemented efficiently for all moduli, matching or improving the efficiency of the best existing schemes for primes while allowing great flexibility in choosing a modulus to optimize cost/performance trade-offs. We also demonstrate that, for a memory-intensive workload on a modern replay-style GPU architecture, prime moduli are not in general the best choices for memory bank and cache set mappings. Applying AMI to set of memory intensive benchmarks eliminates 98% of bank and set conflicts, resulting in an average speedup of 24% over an aggressive baseline system and a 64% average reduction in memory system replays at reasonable implementation cost.

6.1 Solving The Indexing Problem

Traditional indexing techniques lose effectiveness when data locations are not mapped uniformly to the banks or cache locations, leading to bottlenecks that arise from excess demand on a subset of locations. Address mapping is most easily performed by indexing the banks using some number N of low order address bits, i.e. by using a mod(2^N) indexing scheme. However, such schemes are poorly matched to the memory access patterns of many computations, making resource conflicts a significant memory system bottleneck.

Programmers can alleviate such interactions through various means, but these require significant effort, are specialized to specific situations, and are hard to maintain. Hardware solutions include various forms of bit hash indexing, which change but do not eliminate the most common conflicts due to power-of-2 divisors in the index and the memory access patterns. Indexing with non-power-of-2 moduli can alleviate many of these conflicts, but implementing such schemes efficiently in

hardware is difficult. Work in this area has commonly assumed that prime moduli are the best choices to eliminate conflicts since they have the fewest divisors and has thus concentrated on efficient implementations for prime moduli.

The most efficient existing implementations are for moduli of the form $2^N - 1$, some of which are prime (Mersenne primes). For practical systems, only a few possible such choices exist, and they may not be of suitable size. This has led to efforts to find efficient implementations for other primes, including those of the form $2^N + 1$ [18]. While providing more flexibility, prime numbers that are not close to a power of 2 can be difficult to integrate into a power-of-2 based architecture, and indexing implementations for general moduli are relatively expensive. Approaches using prime moduli provide limited flexibility in the choice of modulus, especially in light of integration considerations, and the implementations have not yet proven to be attractive in practice.

In this chapter, we introduce a new indexing scheme called Arbitrary Modulus Indexing (AMI) that can be implemented efficiently for all moduli. AMI matches or improves the efficiency of the best existing schemes for non-power-of-2 indexing while allowing great flexibility in choosing a modulus to optimize cost/performance trade-offs. Our novel implementation of AMI requires as little as 3% of the area of a 32-bit integer multiply unit, less than 0.5% of its power, and just a few gate delays.

As a case study, we evaluate the potential benefits of AMI when applied to a high throughput GPU memory system. Delivering high-bandwidth parallel access to memories requires heavily banking the RAM structures throughout the memory hierarchy. As a result, memory systems of GPUs are at least as reliant on efficient banking techniques as vector supercomputers. In addition, while many parallel algorithms on GPUs benefit from caching, the large numbers of threads put severe pressure on the on-chip caching structures including cache conflict misses. We address two major types of conflicts in the primary memory system of a modern

GPU: (1) *bank conflicts* that arise when multiple threads want to access the same level-1 cache or scratchpad memory bank to obtain different data in the same cycle; and (2) *set conflicts* due to the limited associativity in the cache banks. AMI enables the use of an arbitrary number of cache or scratchpad banks, which reduces many common cross-thread conflict patterns. Given the flexibility of AMI, we examine the performance of a set of memory-intensive benchmarks using a variety of moduli. Surprisingly, we find that the most promising Mersenne prime modulus (31) is not a good choice, and that some of the best moduli are not prime or even odd numbers.

Our results show that minimal additions to the memory system architecture reduce bank conflicts by over 98%, completely eliminating conflicts in 4 of the 5 benchmarks with the highest memory intensity. We also demonstrate that applying AMI to scratchpad banks, L1 cache banks, and L1 sets eliminates 64% of instruction replays, recovering essentially all of the performance lost from conflicts. Most importantly, we show that AMI offers tremendous design flexibility that enables several optimization trade-offs. The area and power overheads are more than offset by gains in performance and reduction in replays. Further, the additional latency required for arbitrary modulus computation is easily hidden in a latency tolerant GPU architecture.

The rest of the chapter is organized as follows. Section 6.2 describes the AMI scheme and its implementation and compares it to existing efficient non-power-of-2 schemes. Section 6.3 provides background on the architecture of throughput processors, discusses where AMI is applied, and details the architectural model used in this paper. Section 6.4 characterizes the behavior of the throughput benchmarks used in this paper, particularly in terms of their conflict behavior, and shows how their behavior varies with choices of index moduli. Section 6.5 quantifies the reduction in conflicts and improvements in performance stemming from AMI. Section 6.6 summarizes our results.

6.2 Arbitrary Modulus Indexing (AMI)

We propose a flexible new approach to efficiently computing indices called Arbitrary Modulus Indexing (AMI). AMI can be implemented more efficiently than any previous non-power-of-2 indexing scheme. Our approach works not just for primes but for any positive integer, albeit with different area/delay costs depending on the exact number used. For about a third of the cases in our case study, AMI adds only a few gate delays to conventional base + index address generation, with area and power equivalent to a few narrow adds. Due to the generality of the approach, it is possible to use a common circuit that can be configured for a variety of moduli, making possible a dynamically configurable indexing modulus for cases where different workloads benefit from different moduli.

AMI has a novel derivation that stems from the original, unmodified hardware implementation of binary reciprocal array multiplication. We then optimize the algorithm and logic design for use in index computation. This approach guarantees an efficient hardware implementation and provides a clear view of how a given circuit implementation would support multiple choices of modulus. Previous methods of computing prime moduli transform the modulus operation into a more efficient mathematical form, but then provide no guidance to efficient hardware implementation. AMI is a general method that subsumes as special cases various mathematical tricks have been developed for special cases of binary division [10, 80]. AMI also has the advantage that both the **div** and **mod** results are produced simultaneously from the same computation, which is important for efficient tag matching and 2-D cache mapping. We will first describe AMI and then use a detailed example to compare it with existing methods.

6.2.1 Efficient Index Implementation

The most efficient general solution to modulus computation to date uses the Chinese Remainder Theorem to convert **mod**(N) to a narrow column of numbers which are then added modulo N. While certainly more efficient than a divide and remainder operation, there is implementation complexity in keeping the sum in **mod**(N) terms. The performance advantage of our method is that it expresses both the **div** and **mod** as a sum of narrow numbers in binary, which can be implemented using a high-performance array adder.

Instead of basing our derivation of **mod** on modular arithmetic, we instead derive it from the original binary **div**/**mod** operation as would occur with binary fixed-point numbers. AMI first multiplies the bits from the address by the reciprocal $(1/N)$ in fixed point. The **div** will be the integer part, while the **mod** will be N times the fractional part of this result. This approach expresses the computation of **div**/**mod** as two integer array multiplies, which on a 1 GHz GPU could complete in less than 2 cycles. Although this solution is already acceptable in terms of latency, it requires unnecessary area and power. From this logical baseline, we derive a far more efficient solution.

A sketch of the derivation of our index computation is as follows. First, we examine the binary form of the reciprocal of N, possibly expressing it as a difference of two numbers to reduce the number of ones. The number and spacing of ones ultimately determines the number and width of the additions needed to multiply by this reciprocal. Second, we derive a transformation to minimize the total number of additions and a hardware mechanism to compactly leverage sequences of infinitely repeating digits. Finally, we take the appropriate bits representing the resulting **div** and demonstrate how to trivially transform them into the **mod** value. The end result can compute a **div** and **mod** that in the best cases use fewer total binary adders than the number of bits in the source address.

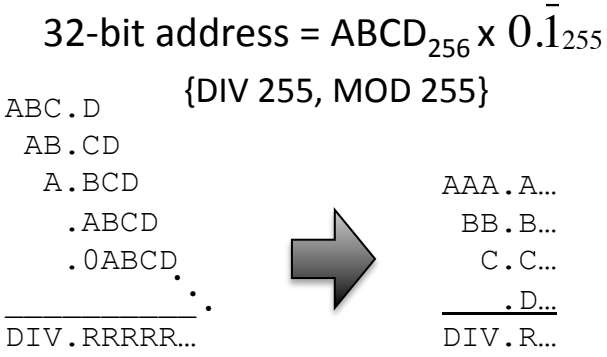


Figure 6.1: Key arithmetic transformation in AMI derivation.

We now explain the detailed algebraic derivation of our approach through a set of examples: First, we recreate the most efficient solutions known for computing **div**/**mod** with moduli of form $2^N \pm 1$. Then, we show how AMI can generalize this to a single circuit computing **div**/**mod** with moduli of the form $2^J \pm 2^K$. Finally, we show how AMI computes **div**/**mod** efficiently for *any* modulus.

Derivation of Efficient $2^N - 1$ div/mod: The goal of an index function that direct maps an address A to S banks or sets is to compute the address pair, $A \Rightarrow (A \textbf{ div}(S), A \textbf{ mod}(S))$. The modulus operator is the critical one, defining the bank or set. The floor divide operator is important in 2-D mappings (bank versus set) and as a tag option.

An efficient **mod**$(2^N - 1)$ function is actually derived from an efficient truncated **div**$(2^N - 1)$ function. For generality of description, assume we will be computing in a logical base, $b = 2^N$, with each logical digit of the number containing N binary digits. We will begin computing $A/(2^N - 1)$ by multiplying A with the constant $1/(2^N - 1)$. This computation is only expressible in base b as an infinitely repeating decimal of the form $0.\overline{1}_b$. This multiplication may be viewed as adding an infinite number of A's together, each shifted over an additional N binary digits, or one logical digit in base $b = 2^N$. Figure 6.1 shows a concrete example of this process in which a 32-bit address A is mapped to 255 ($N = 8$) sets by expressing

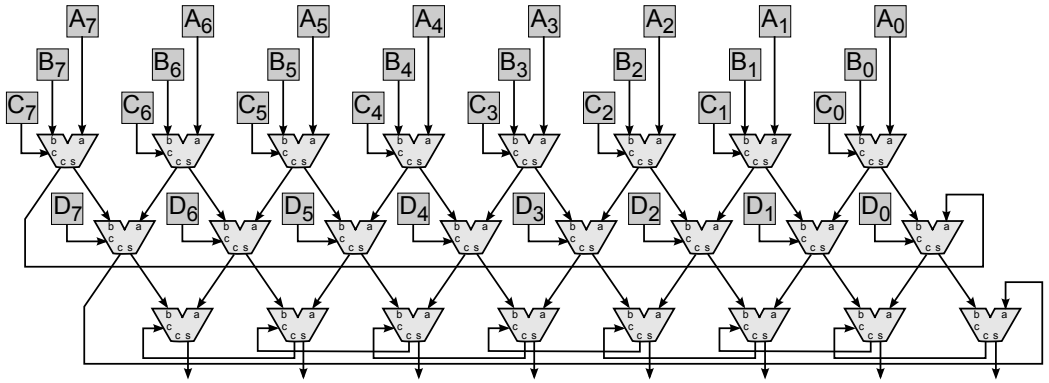


Figure 6.2: 8-bit wide augmented array adder implementation of **mod**(2^N-1) in base 256.

the source address as four base 256 digits, *abcd*. The result of the infinite sum is a three digit number in base 256, representing A **div**$(2^N - 1)$, and an infinite sequence of fractional digits R representing the remainder A **mod**$(2^N - 1)$, again expressed in base 256.

To avoid an infinite number of addends, we apply associativity of addition to transform the sum such that each infinite diagonal represents one addend. Now, instead of adding an infinite number of finite digit numbers, we are adding a finite number (in this case just four) of infinite digit numbers. However, each number just repeats a single digit of A in base 256. As illustrated in Figure 6.1, we can now express the **div**/**mod** operation concisely in this example of four base 256 digits ($A = abcd$) as the sum $(aaa.\overline{a} + bb.\overline{b} + c.\overline{c} + 0.\overline{d}) =$ **div**$(.\overline{R})$. We implement this computation efficiently by augmenting an array adder, where we capture the carry effects of an infinite number of digits by having a wrap-around carry at each stage where the binary carry-out of the remainder digit is fed into the carry-in of the next row. Since each digit is only $N = 8$ bits wide, the wires will be short.

The result of just the circular array addition is the N digit repeating pattern R_b in the infinite sequence of digits representing the **mod** result $0.\overline{R}_b$. To convert this back to base $2^N - 1$, we just multiply $0.\overline{R}_b$ by $2^N - 1$. To do this quickly, we

instead divide by $1/(2^N - 1)$, which we have already shown to be $0.\overline{1}_b$. Dividing $0.\overline{R}_b$ by $0.\overline{1}_b$ is simply R_b, our desired bank (or set) index. An alternative view is we are multiplying by 2^N and then subtracting $0.\overline{R}_b$, removing the repeating digits.

One side effect of preserving infinitely repeating digits (due to the identity $1.0 = 0.\overline{1}$) is that a result that should be in the form {**div**, **mod**} = {**div**, 0} will instead map into {**div**-1, $2^N - 1$}. This is mathematically equivalent, but violates the base range. The standard mathematical answer can be achieved with a simple one-level circuit that checks the **mod** value for all 1's and zeros it out, or by increasing the array width by one binary digit and setting the initial carry-in to 1. However, since we are only using the **mod** value to choose a bank or set, we can omit even this circuitry and just map the SRAM banks as the high $2^N - 1$ values. This example highlights the difference between computing a mathematical **div**/**mod** versus one equally suitable for removing bank conflicts.

Figure 6.2 shows such an array adder implementation using only 24 1-bit full adders to handle 32-bit addresses. Wider addresses need a few more logic levels to handle the extra digits. This operation can easily be folded into a base+offset computation within the same cycle. The rearrangement in Figure 6.1 further shows that we can compute A **div**$(2^N - 1)$ practically for free, as the intermediate sum as each digit is added represents each base b sum digit for the divide. This result matches the best previously reported implementation of **mod**$(2^N - 1)$ by Teng[77] and Dinechin[18].

Derivation of $(2^N + 1)$ div/mod: As before, we start by multiplying A by $1/(2^N + 1)$, but reduce the cost by re-expressing the reciprocal in redundant binary notation $\{-1, 0, +1\}$. This constant is concisely represented in redundant form as $(0.\overline{10}_b - 0.\overline{01}_b) = 0.\overline{11'}_b$, where $1'$ denotes a digit with value -1. Expressing this multiplication and transforming the diagonals in the four digit example yields the sum $(AA'A.\overline{A'A}_b + BB'.\overline{BB'}_b + C.\overline{C'C}_b + 0.\overline{DD'}_b)$. This computation requires

adding sums twice as wide as previously, because the length of repeating digits in the reciprocal is twice is wide. Viewing this sum as $R_1 R_{2b}$, recovering **mod** from the sum requires multiplying by $2^N + 1$, but this is simply $(R_1.R_2 + .R_1)_b$, which is trivial to fold into the array sum. This result matches the best implementation of **mod**$(2^N + 1)$ by Dinechin[18].

Generalizing to $(2^J \pm 2^K)$ div/mod: We extend the prior derivations by expressing **mod**$(2^J \pm 2^K)$ as **mod**$(2^N \pm 1) \times 2^M$. Standard modulo algebra can be re-expressed in binary as taking **mod**$(2^N - 1)$ as before, then prepending the low order M bits from the **div** result. Since our approach already computes **div**, this would be a minimal extension, but we find an even simpler implementation. Take $(A \gg M)$ **mod**$(2^N \pm 1)$, then append the low-order M bits from the original address A. Therefore, the circuit derived for **mod**$(2^N \pm 1)$ also computes **mod**$(2^J \pm 2^K)$, and we can scale any modulus by a factor of 2^N for free. Such **div**/**mod** circuits resemble Figure 6.2 and require fewer total binary adders than bits in the address.

With minor additional multiplexing, we can modify the circuit computing $(2^J + 2^K)$ to also compute $(2^J - 2^K)$ by making the 1's complement optional and choosing just R_1 as the result. In general, wider AMI circuits can emulate narrower circuits with a small amount of configuration.

Generalizing to efficient div/mod of ANY number: A slight generalization of our derivation methodology of **div**/**mod**$(2^N + 1)$ produces an efficient circuit for any modulus M. First, find the repeating bit pattern of the reciprocal $1/M$, whose bit width depends on M. The address A is similarly divided into digits of this width. Then express the reciprocal redundantly. Starting from the right, every consecutive string of 3 or more 1s should be replaced by a difference, e.g. 0111 = 1001'. Multiple strings of 2 or more 1s separated by a single 0 can be expressed as a single large difference with the original zeros becoming -1s, e.g. 011011 = 1001'01'. Each 1 or -1 represents a single layer of logic (an array term), and this approach

guarantees that total array terms remain less than half the number of bits in A, enabling 1-cycle completion for all moduli.

Our case study focuses on moduli between 33 and 61. In this range of 31 choices, 11 have 2 or fewer logic levels per digit, while more than half have 4 or fewer logic levels per digit and widths of 12 bits or less, while 25 have widths of 24 bits or less. Only four moduli, {37, 53, 59, 61} proved relatively inefficient. The total area of the array adder is proportional to the number of bits in the binary representation of A times the number of non-zero binary digits in the repeating reciprocal digit, shown in Figure 6.3.

When the repeating digit width is large, as in 4 of the 31 moduli above, a fallback approach is to dispense with the wrap around carry and use just enough digits to ensure that, with rounding, you will recreate exactly the correct **div** value from the reciprocal multiply. Error analysis indicates that we need to multiply A by a reciprocal truncated to one binary digit more than the width of A, setting the carry-in to 1 for rounding. Even this fallback approach is about 3x more efficient than an integer multiply and can still complete in one cycle. Nonetheless, using AMI gives us enough flexibility to choose more efficient moduli and still achieve full benefits. Two implementations used in this paper are **mod**(62), an array addition of three 5-bit numbers and **mod**(48), an array addition of 12 2-bit numbers, which can be expressed as short parallel sums to arbitrarily reduce gate delay.

6.2.2 Comparison With CRT

While efficient implementations have been derived for the special cases of **div**/**mod**$(2^N \pm 1)$[18, 77], the only algorithm proposed for an arbitrary integer is based on the Chinese Remainder Theorem. We illustrate the competitive advantages of AMI with a concrete example. Sections 6.4 and 6.5 show that for global memory accesses, the best number of banks is 48. This number is also very amenable to interfacing with

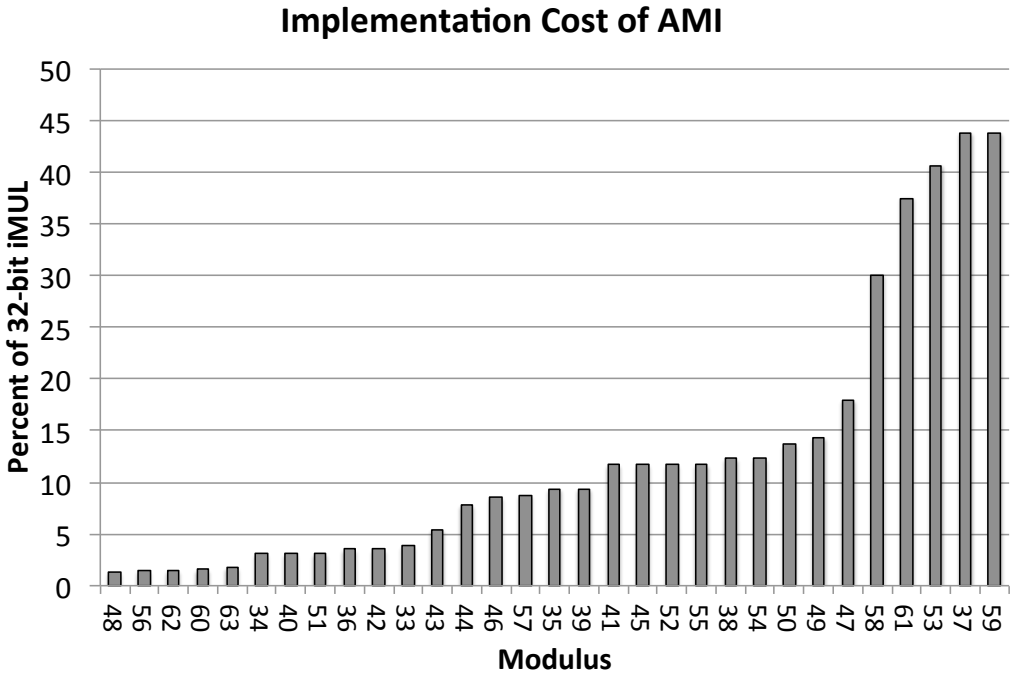


Figure 6.3: AMI Implementation cost versus MOD number.

the rest of the memory system. The AMI derivation shows that since the reciprocal of 48 is $0.000\overline{01}$, the core circuit of the modulus computation of a 24-bit line address will consist of a 2-bit by 12-line array adder with wrap-around carry, with the last four bits of the modulus appended from the low-order bits of the address. This circuit is extremely compact and fast, and requires no special case analysis.

For an efficient implementation based on the Chinese Remainder Theorem (CRT) [77], we need to break the address into digits such that the coefficients are trivial to multiply. We choose the minimal digit size to hold the modulus, base 64. We note that 64^K **mod**(48) is 16, so our coefficients shift the value left by 4-bits. We then compute the sum of 5 terms for each base 64 digit of the address as D_i **mod**(48), sum these values, and then compute a final **mod**. Because there is no circular carry, the sum requires extra bits, and the final **mod** becomes non-trivial. At the very least, this procedure requires roughly 2 cycles to operate: one to

compute the sum of terms, and one to take the **mod** of that sum. However, we can only achieve 2 cycles if there is an efficient way to compute the individual **mod**s of each addend with 48.

Even if a more elaborate analysis of residual arithmetic can separate the problem into a **mod**(3) and **mod**(16) problem, we would end up with a similar 2-bit $\times$ 12 array addition. However, the end sum would still require a full 6 bits as in the original case, so the array area will be closer to that required for a 6-bit $\times$ 12 array addition. Further, such a circuit still does not compute **div** as a byproduct.

Sophisticated CRT implementations (1) may require more elaborate (non-general) derivations to reformulate the problem to eliminate **mod** operations on each term, (2) often result in larger array adders, and (3) in general require an extra cycle to compute the final **mod**. AMI outperforms even the best known implementations using CRT.

6.3 Augmented Replay Architecture

We examine techniques based on AMI for reducing bank conflicts during scratchpad access and bank and set conflicts during L1 cache access. We begin with the detailed Replay Architecture as described in Chapter 5 and augment it as follows. In Figure 5.1, we apply AMI to banks by placing one AMI unit in each lane's address generation unit (for a total of 32) and place one AMI unit at each cache tag array to apply AMI to sets. In addition, we will show to solve index pathologies, we can utilize less of the existing cache tags than before, but will need to add additional banks. We explore two main options — either adding a small number of additional banks with minimal area impact, or doubling the number of scratchpad banks. A final implementation cost is that the all-to-all crossbar connecting lanes to scratchpad banks will need to be widened to support the larger number of banks. We later show that the costs of the additions are negligible in area, power, and added latency,

and will offer significant benefits.

6.4 Benchmarks and Moduli

This section characterizes the memory access behavior of our benchmarks and the sensitivity of this behavior to different numbers of L1 and scratchpad banks.

6.4.1 Memory Conflicts

Bank conflicts tend to be more common and expensive in GPUs than in conventional architectures. In current systems, each bank conflict triggers one or more instruction replays, which costs the processor throughput (issue) slots, lowers total throughput performance, and uses extra power. Replaying memory access instructions multiple times adds large amounts of latency that is difficult for threads and available parallelism to cover. The impact of bank conflicts depends both on memory access intensity and on the bank access pattern.

Memory intensity: Figure 5.2 shows the memory intensity of the benchmarks, which is critical in interpreting the severity of conflicts, as conflicts effectively multiply memory intensity. Five benchmarks, lu, hwt, needle, srad and backprop, have at least 10% memory intensity, which we will group together as *high memory intensive* benchmarks. The next 5 benchmarks have roughly 5% memory intensity, rpes, tpacf, sad, hotspot, and pns, which we classify as *low memory intensive* benchmarks, while the last 3 benchmarks, mri-fhd, mri-q and cp, read from constant memory and apply reductions, so they have essentially no memory intensity. The chart also subdivides the memory references into L1 cache accesses and scratchpad accesses. L1 cache accesses are subject to both set conflicts and bank conflicts, while scratchpad accesses are only subject to bank conflicts.

We focus primarily on the five high memory intensive benchmarks since those will show the greatest effects from changes in the memory system. We found that

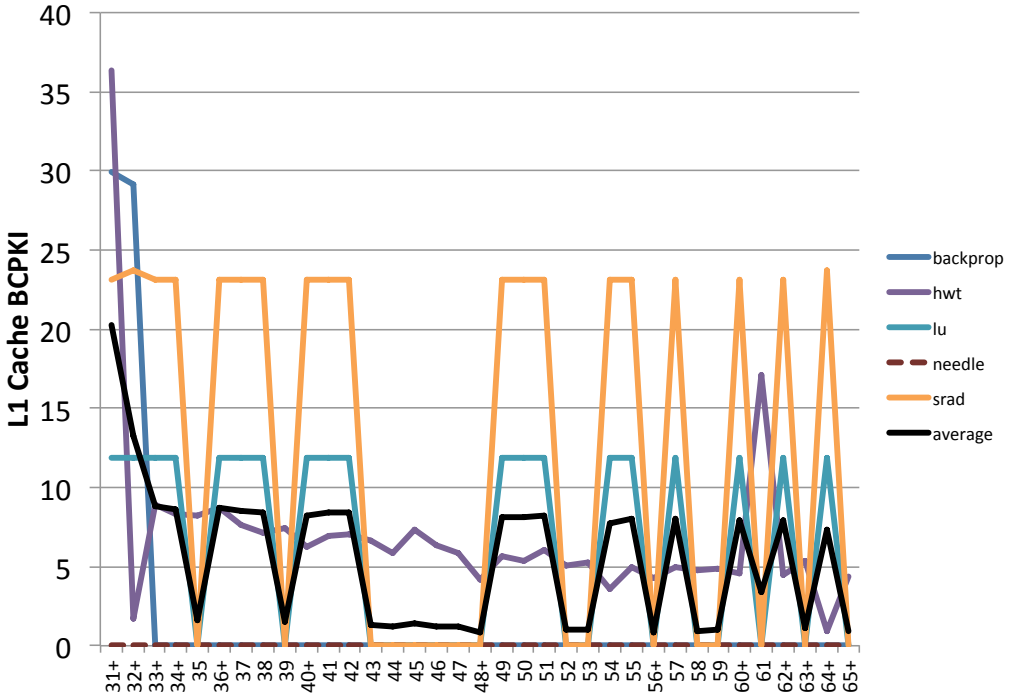


Figure 6.4: Benchmark sensitivity to L1 bank count.

optimizations to the memory system do not degrade performance for the benchmarks with low or no memory intensity (Section 6.5.2).

6.4.2 Sensitivity to Number of Banks

Figures 6.4 and 6.5 show the sensitivity of bank conflicts to bank count for L1 cache and scratchpad accesses, assuming constant L1 or scratchpad capacity, respectively. In contrast to the ideal results above, these and all subsequent results in the paper are produced using a detailed cycle accurate simulation of the memory system. In each graph, the y-axis is the number of bank conflicts per 1000 instructions (BCPKI). The x-axis sweeps the bank count and indicates with a "+" those moduli which are particularly easy to compute, as described in Section 6.2. In general, the number of bank conflicts decreases with increasing bank count. However, the figures show

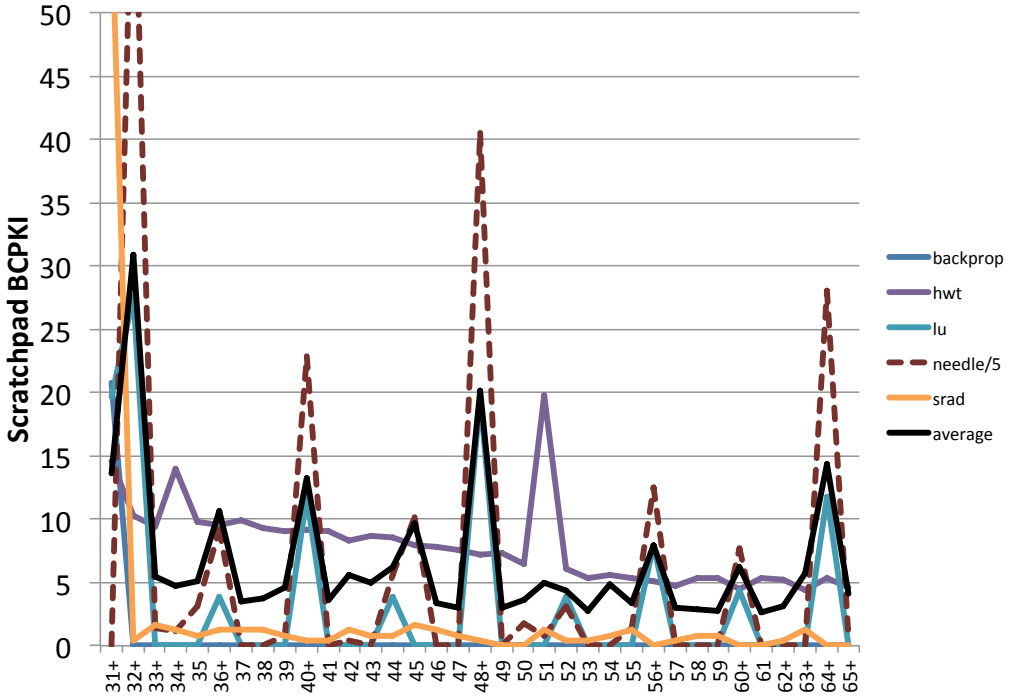


Figure 6.5: Benchmark sensitivity to scratchpad bank count.

significant spikes with scratchpad bank counts at multiples of 8, indicating systematic conflicts between bank count and the memory access stride. The benchmark hwt has irregular memory access patterns, leading to fewer conflicts than the other more strided applications. The L1 cache is less sensitive to bank count and its bank conflicts often spike at different bank counts than the scratchpad.

These figures show that the prime numbers are not necessarily the best choice for bank counts. For the L1 cache, the bank conflicts fall to zero at a bank count of 48, which not only is not prime but also is not an odd number, and many even numbers are good candidates for both L1 and scratchpad.

6.5 Performance Results

As indicated above, this section uses a detailed, cycle-accurate simulation to evaluate the performance of AMI in the context of a full secondary memory system.

Bank and moduli configurations: AMI indexing can be disabled or configured to index by any modulus choice of sets or banks *less than* the existing physical sets or banks. For sets, using less than the total number of cache lines is a net benefit. For banking, in the context of a GPU, we cannot constructively index less than 32 banks without (in most cases) increasing bank conflicts, so altering the physical scratchpad/L1 is necessary.

We evaluate two different physical scratchpad augmentations. In one case, we simply add a 33rd or 34th SRAM bank, increasing area and leakage power by 3-6%, but having minimal impact on the crossbar networks. This enables indexing modulo 32 to 34 within the scratchpad or L1 cache. The L1 and scratchpad are disjoint, so they can freely use different index moduli. The second approach is to keep the cache size the same, but convert it to 64 half-sized banks, doubling the crossbar size from 32:32 to 32:64. The cost of this change is negligible in power and area (Section 6.5.4), and it allows indexing with any moduli up to 64 for both the L1 cache and scratchpad. For the L1 cache, cache lines are always maintained as 128 consecutive bytes, regardless of the number of banks, to preserve compatibility with the rest of the memory system. In the following sections, the term "bank count" may be used interchangeably with the indexing modulus and does not represent the underlying physical bank count.

While we present performance sweeps in Section 6.5.3, in the next section we highlight a few sample moduli to illustrate the benefits of AMI at different physical bank configurations. We chose 34 and 62 as two extremes for scratchpad access, and 48 in the middle for L1 cache access. None of these moduli have been studied before, and all are easy to compute (Figure 6.3) and have robust performance across

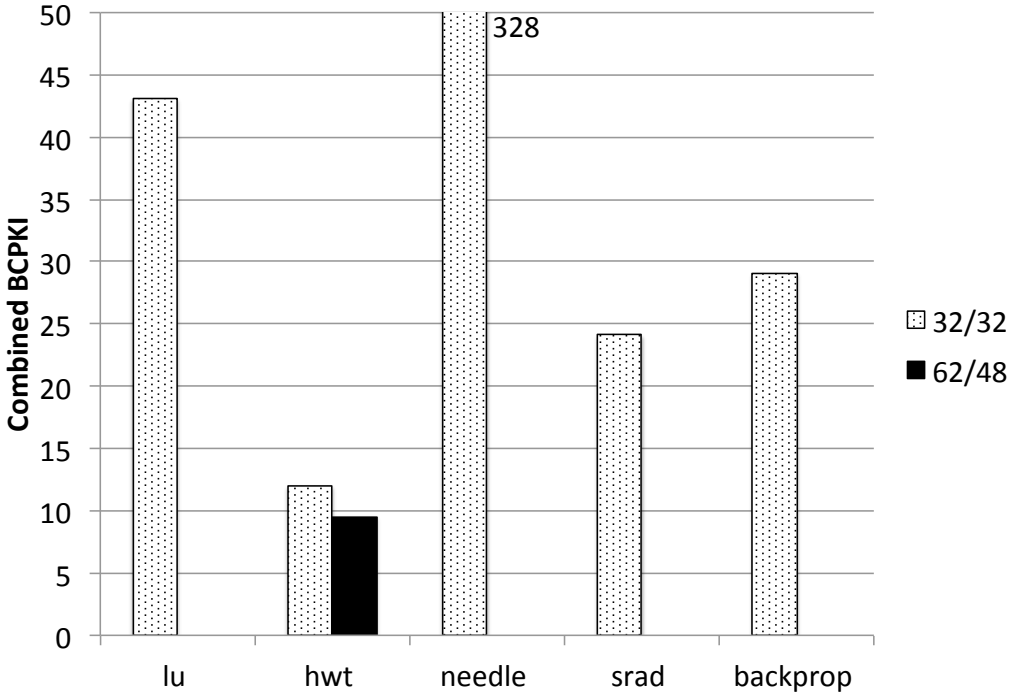


Figure 6.6: Total bank conflicts per thousand instructions for baseline vs AMI-Banking.

all benchmarks, achieving near optimum in both performance and power.

6.5.1 AMI Bank Conflict Reduction

As shown in Figure 6.5, for the base case of 32 banks, four of the five memory intensive benchmarks have severe bank conflict issues of 20% or greater. One benchmark, hwt, only has moderate conflicts of 12%, due to irregular memory access patterns, which tend to be less pathological. Figure 6.6 demonstrates that applying AMI-banking with a single, static number of banks across all benchmarks and kernels reduces 98% of bank conflicts. In the figure, the gray bar represents the baseline architecture with 32 banks for the scratchpad and 32 banks for the L1 cache. The AMI enhanced architecture uses 62 banks for the scratchpad and 48 banks for the L1 cache. In 4 of the 5 benchmarks with the most severe conflict issues (backprop, lu,

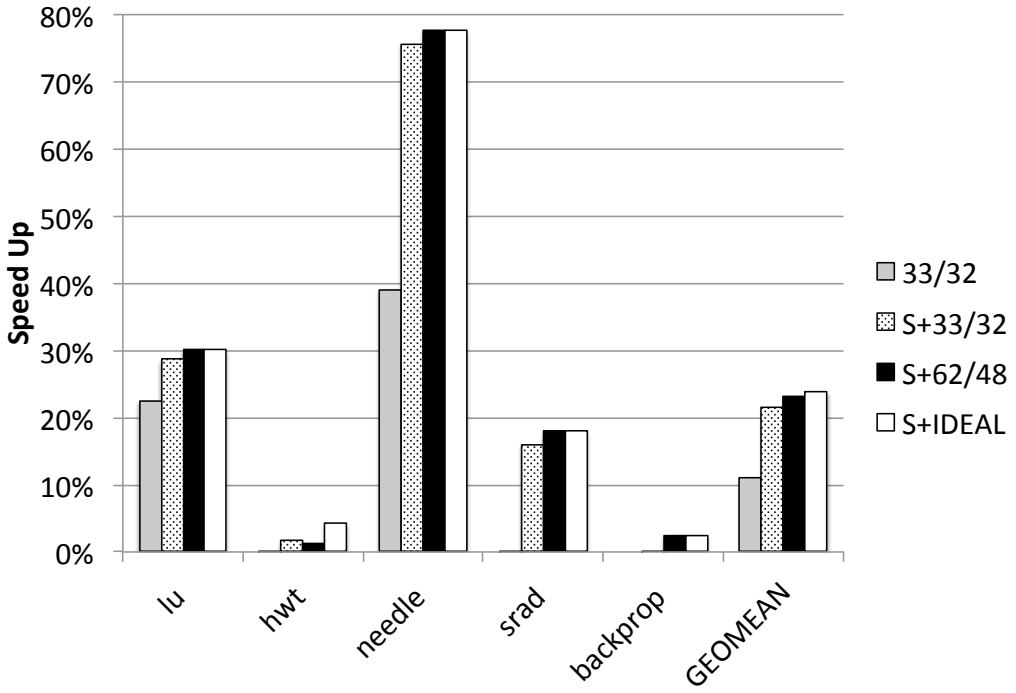


Figure 6.7: AMI speedup over base case of 32/32 banks and 2 tag arrays.

needle, and srad), conflicts are reduced to zero, without requiring any programmer intervention. The one outlier, hwt, has an irregular memory access pattern, making it less amenable to AMI improvements, but also resulting in fewer initial conflicts.

6.5.2 AMI Performance

Speedup: Figure 6.7 shows the increase in throughput performance resulting from bank and set conflict reduction. Our notation in the figures is (#SP banks/#L1 banks), with S+ denoting the addition of AMI-sets. Applying AMI to scratchpad banks and using the cheapest solution of 33 banks yields a geometric mean 11% speedup, or nearly half the ideal speedup for no bank conflicts. Applying AMI-sets to the L1 cache, an extremely low cost optimization, increases aggregate speedup to 21.4%. For execution speed, supporting the two lowest cost AMI implementations (S+33/32) already achieves 90% of ideal speedup. Finally, applying a more aggres-

sive choice of static AMI bank numbers to both the scratchpad and L1 cache leads to a 23.1% geometric mean speedup, or 97% of ideal speedup. This performance from static bank counts suggests that dynamic AMI mappings per benchmark are not likely to be worth the additional complexity, at least for this workload.

Three benchmarks (lu, needle, and srad) see large throughput performance gains of 18%–78%, while hwt and backprop see moderate performance gains of 2.4–4.2%. Irregular memory access patterns and a high ratio of load-dependent instructions prevent hwt from seeing a large increase. The limited effects in backprop are due to cache misses that mask the additional latency caused by local replays. These overall improvements are very significant for a throughput architecture, which by its very nature is designed to mask high latency through massive task level parallelism, and they are achieved over a very aggressive base case, with very limited application of AMI.

Finally, we note that there were no deleterious effects for the 8 remaining low memory intensity benchmarks. Those benchmarks showed a geometric mean speedup of 0.94%, the largest winner being hotspot, with a 4.92% speedup, and sad and tpacf, with 1.66% and 1.33% speedups, respectively. Only one benchmark, pns, saw a negligible (0.06%) slow down, which can be avoided by switching off AMI.

Replay and Energy Reduction: In Chapter 5, we showed that benchmarks experienced a large number of bank conflicts, and that for every bank conflict between a single warp and a single bank, the entire warp instruction would have to be replayed. Figure 6.8 shows total replays per thousand instructions (RPKI). Comparing with Figure 6.6, we see that bank conflicts can increase replays by an order of magnitude since the relatively long pipelines of throughput architectures lead to a large number of instructions in flight.

Overall, 4 of the 5 memory intensive benchmarks show absolute reduction in replays of nearly 20% or more, while 3 have replays reduced by 50% or more. Hwt

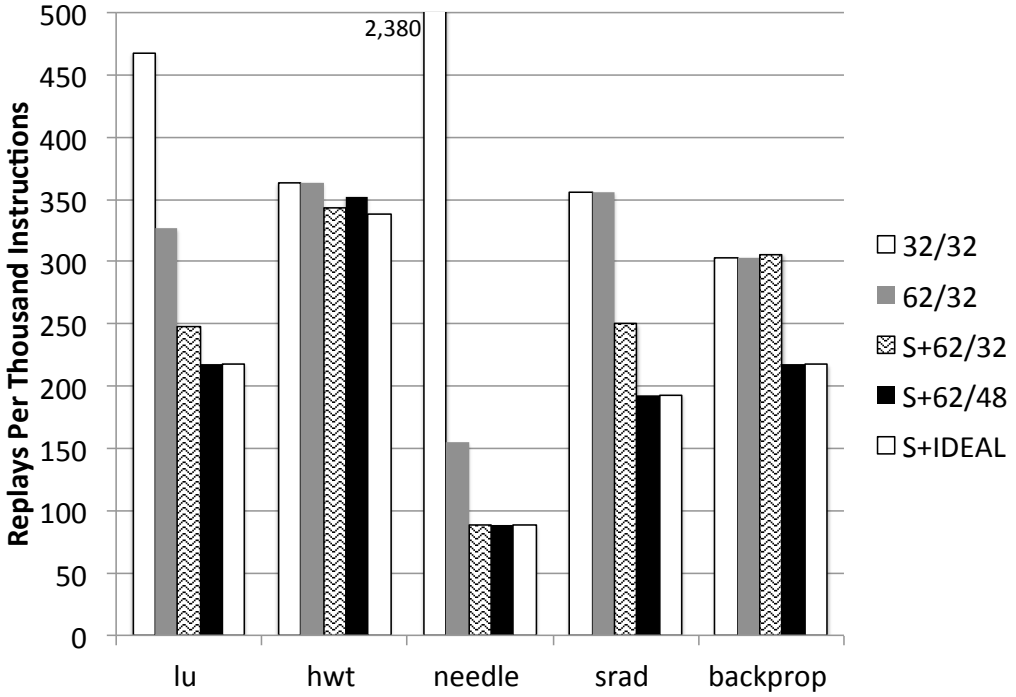


Figure 6.8: Reduction in replays from applying AMI.

only showed a modest reduction in replays of about 7%, owing both to its limited reduction in bank conflicts from AMI due to its irregular memory access pattern and the low number of instructions associated with each replay. Two of the benchmarks have substantial replay reductions from just applying AMI to scratchpad banks and four of the benchmarks benefited from applying AMI to L1 cache sets. Finally, 3 of the benchmarks had significant improvements from applying AMI to L1 banks, despite L1 banks having minimal impact on execution speed. All benchmarks were able to achieve nearly ideal (conflict-free) levels of replay reduction with appropriate use of AMI.

Figure 6.9 shows the fraction of total instructions executed by (S+62/48) relative to the baseline 32/32 architecture. The reduction in executed instructions stems directly from the set and bank conflict replays eliminated by AMI. The

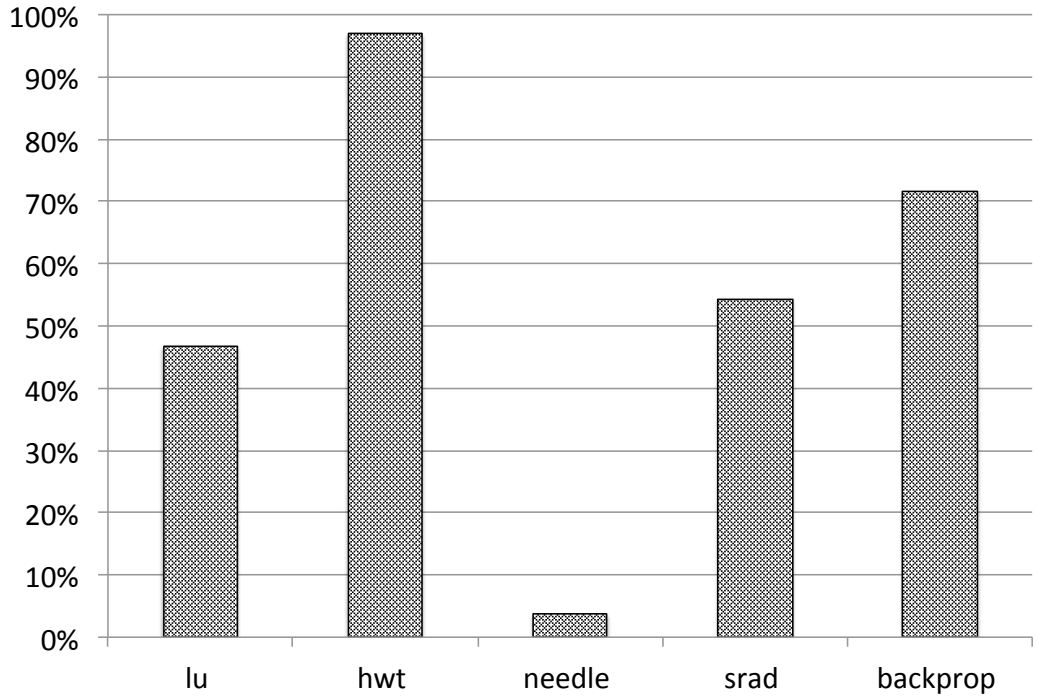


Figure 6.9: Fraction of total instructions executed relative to baseline 32/32.

benchmark needle sees the most impressive improvement, reducing its executed instruction count by about 95%. Instructions executed can be used as a first order proxy for power consumption, indicating that any power costs for AMI are offset by the benefits from reduced replays.

6.5.3 Performance Sensitivity to Bank Count

Figure 6.10 shows the variation in speedup and replays when varying scratchpad bank count and holding the rest of the system constant. Bank counts with a + represent those with the simplest AMI index computation. The graph shows significant performance benefits (up to 17% speedup or 62% replay reduction) from just altering the number of scratchpad banks. About a dozen different bank counts are comparable in achieving most of the performance benefits in both speedup and RPKI. This effect is important, first, because it only requires a small number of

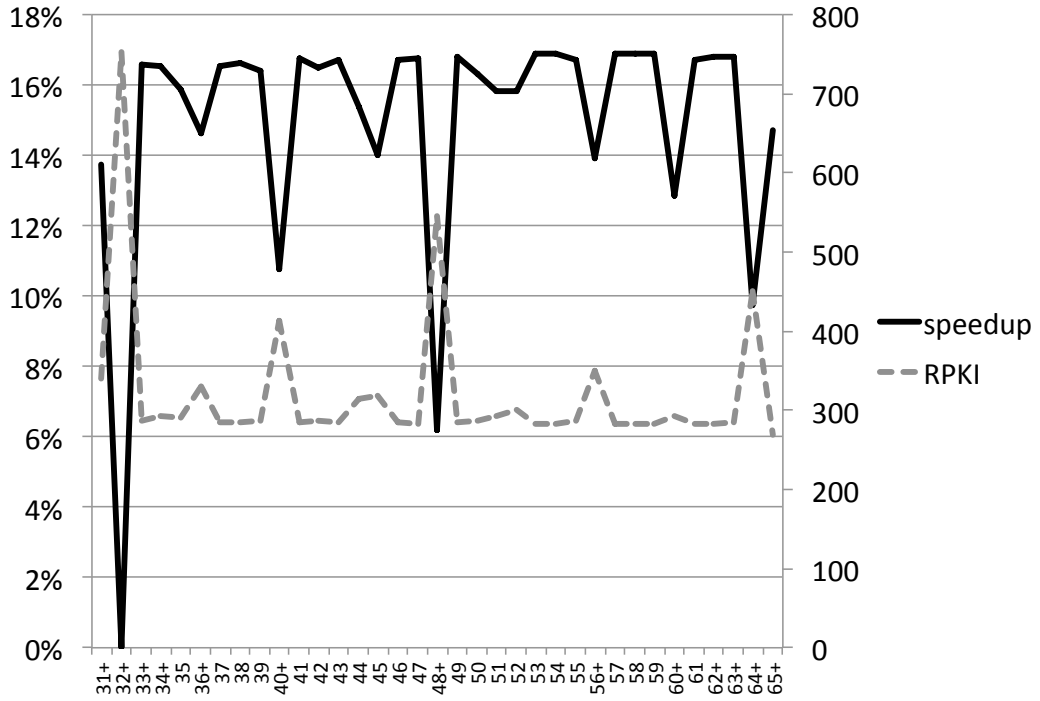


Figure 6.10: Speedup and RPKI vs number of scratchpad banks.

indexing options to optimize all benchmarks, so a single, static choice can achieve near ideal performance over all benchmarks. Second, ease of index computation is not the only integration consideration in a system, and certain moduli may be favored for architectural reasons.

While all benchmarks benefit from having a bank modulus other than 32, there are variations in performance with moduli, and different benchmarks (and kernels) see better results at different moduli. Furthermore, optimal moduli differ between scratchpad and L1-cache, and differ for throughput performance vs energy savings (L1 sensitivities in Figure 6.4).

AMI allows hardware designers to make clean tradeoffs between index cost, networking cost, average speedup, average energy savings, and robustness across benchmarks, at a relatively insignificant cost in power, area, or complexity.

6.5.4 Integration costs

AMI circuits: We need to add 32 AMI circuits per core (one for each ALU lane), and two for the tag arrays. Active and leakage power are highly dependent on the exact VLSI implementation of a circuit used. As a conservative estimate of power, we extrapolate power and area figures from standard cells used in [31], which were based on 32nm BSIM-4 predictive technology models (PTM). The AMI circuit is very similar to a ripple adder with the same number of bits as the address in the simplest case, or double that in the next simplest case. Linear extrapolation of the 3-bit adder cell from [31] leads in the worse case to 222fJ energy per DIV/MOD computation and an area of 8.6um^2, representing overheads of 0.5% in active power and 0.1% in leakage power compared to the 32-bit FMAD in each lane. Performing an AMI index computation for each lane for every global and scratchpad memory access results in an average addition of 0.68 milliwatts active power, ranging from 0 to 2.26 milliwatts for the highest benchmark intensity. The delay of AMI circuits is less than one cycle, but even if AMI forced adding a cycle to the pipeline, it would have no visible impact on performance, because GPU scratchpad and cache latency is tens of cycles and there is ample time to run AMI in parallel, off the critical path.

L1 sets: A single AMI circuit is used at each tag array to restrict tag lookup to less than the total amount. The cost of an AMI circuit is negligible in power, area, and delay, and the reduction in set conflicts more than offsets having fewer cache lines. However, because a set location now depends on more of the address, additional tag bits are needed, the amount depending on the modulus. In the worse case, storing an extra 8 bits per tag entry only increases scratchpad area by half a percent. Our implementation retains the notion of 128-byte cache lines, requiring no changes to the TLB or L2 cache interface. Mapping a given cache line to a physical location involves rotation, which in our case is provided by the existing crossbar network.

Scratchpad/L1 banks: To implement true scatter gather, a large crossbar connects 32 ALU lanes to the default 32 scratchpad/L1 banks. Implementing AMI involves changing the crossbar to 32×N, and possibly adding additional SRAM banks. For a choice of 33, adding one more SRAM bank would be the most sensible, while a choice of 62 would likely double the bank count and crossbar size. When adding banks to the L1/scratchpad, active access power is actually reduced, because the same number of bank accesses occur on smaller banks. Leakage power increases slightly due to the overhead of control circuitry as banks become smaller. Adding a single bank only reflects a 3% increase in area, while doubling the number of banks also only requires a minimal increase in area and leakage power due to increased overhead per bank.

Crossbar: The largest change in the crossbar network moves from 32×32 to 32×64, doubling the number of access muxes and wire links. We simulated link by link crossbar traversals for all benchmarks. Our simulation modeled a monolithic rectangular crossbar in which each link activation was measured. The average data traversal length was a little less than the average manhattan distance due to lane zero being used with greater frequency. We then conservatively estimated total power as roughly double the cost of simulated wire energy of that configuration to account for active logic energy. Linearly estimating the area of the SRAM banks from [31] and using wire capacitance and minimum spacing for lanes from the ITRS Roadmap [37], the active wire power at 64 banks ranges from 0.6 milliwatts to 1.9 milliwatts. Doubling that as a total estimate yields 1.2 milliwatts, which varies linearly with the number of banks chosen. In comparison, the energy to access the SRAM banks is an order of magnitude greater, at 116 milliwatts, and access to registers in the core pipeline is even higher. Thus the extra power needed for AMI banking is a small fraction of core power.

6.6 Summary

In this chapter, we introduce a new scheme for address mapping using Arbitrary Modulus Indexing (AMI)[22]. We show that for non-power-of-2 indexing, our scheme can be implemented as or more efficiently than previous schemes, even for the most competitive case of Mersenne prime indexing. We then show that set and bank conflicts in the primary memory system of a replay-style throughput architecture have a deleterious effect on system performance caused by the alignment of the bank count with natural strided memory access patterns across the threads in a warp, similar to previously known effects for vector supercomputers. We show how to use AMI to obtain significant gains in performance and power efficiency on such an architecture. As discussed in Chapter 3, we again see the importance of single thread latency to overall throughput performance, and again see major performance opportunities in the primary on-chip memory system. In this case, by reducing conflicts, we made caches more efficient, and had dramatic improvements in applications that had high memory intensity and regular access patterns.

As described in Chapter 3, reducing conflicts reduces effective hit latency and increases throughput performance. The resulting system is simple to implement and provides robust benefits across all of our benchmarks, on average eliminating 98% of bank conflicts, and completely eliminating bank conflicts on 4 of the 5 benchmarks with the most serious conflict issues, with no performance detriments for benchmarks with low memory intensity. Applying AMI to scratchpad banks, L1 banks, and L1 sets achieved 98% of ideal performance, resulting in a geometric mean speedup of 24% across the 5 most memory intensive benchmarks, significant for an aggressive baseline architecture designed to mask latency with explicit TLP. AMI also resulted in a 64% reduction in instruction replays. The cost of our implementation is just a few percent of the area and active power of a 32-bit array multiplier and adds just a few gate delays to the pipeline, while worst case power increases due to sort

networks and cross bars is under 2 milliwatts.

AMI provides great design flexibility, enabling trade-offs in throughput performance, power reduction, and implementation cost. Our scheme works for all moduli, and relatively efficient implementations of our scheme apply to a large number of moduli. This opens up the possibility of separating the indexing modulus from the physical implementation of a cache or memory banks and tuning the modulus for best performance on a given workload. In fact, it is feasible to implement a single mapping circuit that can be dynamically tailored to moduli best suited for individual applications, although our performance results indicate this is unnecessary for the benchmarks we used. AMI benefits are orthogonal to, and can enhance other indexing schemes designed to work on power of 2 sized caches. Given this flexibility, we expect schemes based on AMI to be useful for other aspects of the memory systems of various types of architectures.

Chapter 7

Conclusions and Future Work

The goal of this dissertation was to examine the properties of throughput architectures and attempt to improve parallel efficiency at the hardware/software interface in a way that was both general and transparent to the programmer. We have demonstrated that modern parallel CPUs are indeed operating in a throughput regime, and classical approaches to parallelism are not the best fit. We succeeded in creating a detailed mathematical model that demonstrated counter-intuitive and previously unknown tradeoffs in throughput performance in the context of shared on-chip memory systems. We then used this model to successfully develop two general purpose throughput optimizations — one in hardware and one in software — that satisfied all of the criteria we outlined.

This dissertation began by defining a throughput architecture as one that trades off the number of hardware cores or threads with shared on-chip resources in a way that may reduce single thread performance but boosts total chip performance. A mathematical analysis illustrated the degree to which on-chip memory and off-chip bandwidth can impact throughput performance as well as the fragility of such a system and the dependency on application and cache characteristics.

To examine coarse grained throughput architectures, we then studied a full

scale legacy supercomputing application in-situ and discovered severe intra-chip scaling issues across multiple cores due to conflicts throughout the shared memory system. We also observed that any perturbation of the memory system lasted an order of magnitude longer than on uniprocessor chips, ruling out standard statistical sampling based performance analysis tools. After developing a toolset for multicore performance analysis, we demonstrated that an application's working set size can be dramatically reduced by local loop transformations, improving throughput and scalability.

For fine grained parallelism, we studied an abstract Replay Architecture using a custom simulator and characterized the Rodinia and Parboil benchmark suites, finding the largest single performance bottleneck was caused by bank conflicts in the scratch pads. By applying a new technique we developed called Arbitrary Modulus Indexing, we eliminated all bank and set conflicts in both the scratchpad and L1 caches, improving throughput performance and parallel efficiency.

The main conclusion of this dissertation is that the primary on-chip memory system is as important to throughput performance as any other major area of study, but has yet to be adequately studied due to the limitations on current simulation tools. We show that while multicore and multithreaded optimizations seem complex, localized solutions are possible, especially when leveraging hardware/software co-design.

7.1 Dissertation Contributions

As discussed in more depth in Chapter 1, this dissertation has performed studies that dispelled common wisdom in many areas. To summarize the largest contributions of this dissertation:

- A Mathematical Model of Throughput Performance: We examined the notion of what throughput performance and caching performance means in the con-

text of a massively multithreaded architecture, bringing such understanding to the next level of detail.

- Multicore Optimization: We demonstrated that supercomputing applications have intrachip scaling issues, and that different performance counters, measurement systems, and time scales were needed to evaluate their performance. We developed a new approach and performance analysis library to measure performance counters, and a local software optimization called "Loop Fission" that improved scalability by transforming loops to reduce local working set sizes and bandwidth requirements.

- Custom Simulator: We build the first simulator for Replay Architectures that accurately modeled the long latency and primary on-chip memory system of Replay Architectures, and that was fast enough for easy simulation of large scale runs.

- Application Characteristics: We found many surprising characteristics in fine-grained throughput applications. We found that many had high memory intensity, with memory patterns that were very amendable to caching, and that those with regular memory access patterns were more challenging to the memory system than those with irregular access patterns.

- Arbitrary Modulus Indexing: We developed an efficient technique for implementing the DIV and MOD of an arbitrary constant integer, finally generalizing known mathematical results for Mersenne Primes. This technique was applied to the L1 cache tags, and L1 and scratchpad banks of a Replay architecture where it eliminated both bank conflicts and cache conflict misses, boosting throughput performance and parallel efficiency. However, similar techniques can be trivially applied to L2 and last level caches, on-chip networks and routers, memory controllers, TLBs, MSHRs, and more.

7.2 Future Directions

This dissertation has opened up several different areas that merit future attention.

We have demonstrated the importance of the primary memory system to throughput performance, while highlighting that the most popular academic simulator does not model this. As a result, it is worth revisiting many studies on thread scheduling and cache policies that were done assuming both L1 caches and scratch pads can be accessed in a single cycle. Portions of our simulator have already been ported to GPGPU-Sim by Calvin Lin's group at the University of Texas at Austin, and this process will continue.

Having shown the potential throughput performance advantages of better cache and scratchpad utilization, we have barely scratched the surface of hardware and software evolution that could result in higher hit rates and lower contention. Significant research areas are still open in the areas of MSHR design, scalar register files, low latency "L0" caches, cache bypassing, and new cache protocols. Having exposed the nature of a Replay Architecture, academic research can now explore such architectures and look at variations and tradeoffs across multiple throughput designs.

Our fine-grained architecture studies were based on old fashioned benchmark kernels. It is important to look into more modern, more sophisticated applications that may have different characteristics and needs. Additionally, many applications have minimal data reuse, and there are many more ways to support parallelism than explicit threads. There are completely new design spaces that should be explored in which items of work are not explicitly tied to threads, and where lighter weight task management hardware is responsible for gathering task data and distributing it to hardware cores for execution.

Finally, the applications of AMI have only begun to be explored. AMI can be trivially extended to more of the memory system - register banks, MSHR files,

TLB caches, L2 and L3 tag arrays and banks, memory banks, etc. AMI has already been implemented for Network On Chip algorithms, and there have been repeated requests to create an open source library of RTL implementations of AMI circuits. In almost every aspect of the chip, there are places in which data must reside on a power-of-2 number of locations and easily located there without pathological conflicts, and all of these can be reconsidered in the new context of AMI. There have been multiple requests from both industry and academics for an open source RTL library of AMI circuits. This could be more easily achieved with software that auto generates RTL based on user requirements, and could even be given a web front end for use.

7.3 Final Thoughts

This dissertation has explored the fundamental throughput tradeoffs of cores and threads versus on-chip memory and off-chip bandwidth. Along the way, we have developed new techniques for analyzing and optimizing hardware and software on these architectures. However, now that we have entered the era of explicit parallelism on a chip, I feel there are totally new types of throughput architectures that can be explored, ones that are not extrapolations of conventional processors, but which more efficiently distribute and execute parallel tasks in a way that is nearly invisible to the programmer. The key is to observe the problem at the highest sensible level, rather than focusing on how to implement low level pieces.

I also feel the architectural community would greatly benefit from a new approach to simulation infrastructure. We are entering an era where incremental architecture tweaks are no longer enough, and existing simulation tools are (1) over specified and difficult to change, and (2) monolithic to the point where people must take them on faith, assume they are debugged, and gloss over the details of implementation. I feel a more helpful approach is to create a standard object model

for interconnecting simulation pieces, and then to have small, individual simulation pieces be developed and open sourced. These individual pieces would be easier to study, debug, and vary than current monolithic simulators. It would be easier to supply different versions of a given architectural piece, at different levels of detail or application. Additional open source projects should aim to use the same framework to develop tools that help monitor and analyze performance, track statistics, and output and visualize analysis. We need a similar approach to the compiler front end. Despite decades of compiler work and the LLVM compiler reboot, it is still not easy to do what every architect needs to do to provide custom instructions and compiler optimizations into the system. These are first rate goals, and the end result will make student's lives easier and help to push truly novel designs in a post Moore's Law world.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, April 2010.

[2] A. Agarwal and S. D. Pudar. Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-mapped Caches. In *Proceedings of the International Symposium on Computer Architecture*, pages 179–190, May 1993.

[3] *BIOS and Kernel Developers Guide (BKDG) For AMD Family 10h Processors, http://developer.amd.com/documentation/guides/pages/default.aspx*, rev 3.48 edition, April 2010. Publication 31116.

[4] T. Austin, D. Pnevmatikatos, and G. Sohi. Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the International Symposium on Computer Architecture*, pages 369–380, June 1995.

[5] D. F. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. In *ACM Computing Surveys*, volume 26, pages 345–420, December 1994.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the*

International Symposium on Performance Analysis of Systems and Software, pages 163–174, April 2009.

[7] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[8] B. Bershad, D. Lee, T. Romer, and J. Chen. Avoiding Conflict Misses Dynamically in Large Direct-mapped Caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems*, pages 158–170, October 1994.

[9] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-10)*, November 2010.

[10] M. Calhoun. On the Binary Decimal Expansion of the Reciprocal Prime's, 2010.

[11] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi. User-defined Distributions and Layouts in Chapel: Philosophy and Framework. In *Proceedings of the USENIX Conference on Hot Topics in Parallelism*, May 2010.

[12] 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 *International Symposium on Workload Characterization*, pages 44–54, October 2009.

[13] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, December 2010.

[14] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *International Conference on High Performance Networking and Computing (Supercomputing)*, November 2011.

[15] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang. Memory latency reduction via thread throttling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 53–64, Washington, DC, USA, 2010. IEEE Computer Society.

[16] T. Chilimbi, M. Hill, and J. Larus. Making Pointer-based Data Structures Cache Conscious. *IEEE Computer*, 33(12):67–74, December 2000.

[17] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Autotuning on State-of-the-art Multicore Architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-08)*, November 2008.

[18] B. D. de Dinechin. A Ultra Fast Euclidean Division Algorithm for Prime Memory Systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 56–65, November 1991.

[19] J. Diamond, M. Burtscher, J. McCalpin, B.-D. Kim, S. Keckler, and J. Browne. Evaluation and Optimization of Multicore Performance Bottlenecks in Supercomputing Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2011.

[20] J. Diamond, B.-D. Kim, M. Burtscher, S. Keckler, K. Pingali, and J. Browne. Multicore Optimization for Ranger. In *Teragrid Conference*, June 2009.

[21] J. Diamond, J. D. McCalpin, M. Burtscher, B.-D. Kim, S. W. Keckler, and J. C. Browne. Making sense of performance counter measurements on super-

computing applications. Technical Report TR-10-25, University of Texas at Austin, Department of Computer Science, July 2010.

[22] J. R. Diamond, D. S. Fussell, and S. W. Keckler. Arbitrary modulus indexing. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 140–152. IEEE, 2014.

[23] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, September 2010.

[24] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, 3:3–10, February 2007.

[25] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a Vector Extension to the Alpha Architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 281–292, May 2002.

[26] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-core Systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 169–180, February 2011.

[27] Q. S. Gao. The Chinese Remainder Theorem and the Prime Memory System. In *Proceedings of the International Symposium on Computer Architecture*, pages 337–340, May 1993.

[28] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient Mechanisms for Managing Thread Context

in Throughput Processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 235–246, June 2011.

[29] T. Givargis. Improved Indexing for Cache Miss Reduction in Embedded Systems. In *Design Automation Conference*, pages 875–880, June 2003.

[30] S. L. Graham, P. B. Kessler, and M. K. Mckusick. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[31] X. Guo, E. Ipek, and T. Soyata. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. In *Proceedings of the International Symposium on Computer Architecture*, pages 371–382, June 2010.

[32] Z. Guz, E. Bolotin, I. Keidar, avinoam kolodny, A. Mendelson, and U. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letters*, 2008.

[33] Z. Guz, O. Itzhak, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. Threads vs. caches: Modeling the behavior of parallel workloads. In *28th International Conference on Computer Design, ICCD 2010, 3-6 October 2010, Amsterdam, The Netherlands, Proceedings*, pages 274–281. IEEE, 2010.

[34] J. Hack, B. Boville, B. Briegleb, J. Kiehl, P. Rasch, and D. Williamson. Description of the NCAR Community Climate Model (CCM2). Technical report, NCAR, Boulder, Colorado, 1993. NCAR Tech. Note TN-382.

[35] E. Hallnor and S. Reinhardt. A Fully Associative Software-managed Cache Design. In *Proceedings of the International Symposium on Computer Architecture*, pages 107–116, May 2000.

[36] *System Programming Guide, Part 2.* Intel document 253669.

[37] International Technology Roadmap for Semiconductors (ITRS), 2011 Edition.

[38] R. Jayaseelan, A. Bhowmik, and R. D. C. Ju. Investigating the Impact of Code Generation on Performance Characteristics of Integer Programs. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architecture (INTERACT)*, pages 1–8, 2010.

[39] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the International Symposium on Computer Architecture*, pages 332–343, June 2013.

[40] D. R. Johnson, M. R. Johnson, J. H. Kelm, W. Tuohy, S. S. Lumetta, and S. J. Patel. Rigel: A 1, 024-core single-chip accelerator architecture. *IEEE Micro*, 31(4):30–41, 2011.

[41] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *Micro, IEEE*, 31(5):7 –17, sept.-oct. 2011.

[42] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA*, pages 140–151, 2009.

[43] A. Kerr, G. Diamos, and S. Yalamanchili. GPUOcelot - A Binary Translator Framework for PTX, October 2009.

[44] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using Prime numbers for Cache Indexing to Eliminate Conflict Misses. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 288–299, February 2004.

[45] M. Kharbutli, Y. Solihin, and J. Lee. Eliminating Conflict Misses using Prime Number-based Cache Indexing. *IEEE Transactions on Computers*, 54(5):573–586, May 2005.

[46] C. Kim, D. Burger, and S. W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, pages 99–107, 2003.

[47] P. Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical Report TR-2008-13, University of Notre Dame, 2008.

[48] D. Lawrie and C. Vora. The Prime Memory System for Array Access. *IEEE Transactions on Computers*, 100(5):435–442, May 1982.

[49] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 213–224. IEEE, 2010.

[50] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. Aamodt, and V. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture*, pages 487–498, June 2013.

[51] Longhorn User's Guide, http://services.tacc.utexas.edu/ index.php/longhorn-user-guide.

[52] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, F. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, June 2005.

[53] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2010. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[54] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[55] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130, April 2003.

[56] P. Micikevicius. GPU Performance Analysis and Optimization. *GPU Technology Conference*, `http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis.pdf`, May 2012.

[57] A. L. Minkin, S. J. Heinrich, R. Selvanesan, C. McCarver, S. G. Carlton, M. Y. Siu, Y. Y. Tang, and R. J. Stoll. Cache Miss Processing Using a Defer/Replay Mechanism, September 2012.

[58] MPI Profiler, http://mpip.sourceforge.net.

[59] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. We Have It Easy, But Do We Have It Right? In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7, April 2008.

[60] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Sym-*

posium on Microarchitecture, MICRO-44 '11, pages 308–317, New York, NY, USA, 2011. ACM.

[61] NSF 0605: The High-Performance Computing Challenge Benchmarks, version 2.0.

[62] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[63] L. Oliker, J. Shalf, and K. Yelick. Optimization of a Lattice Boltzmann Computation on State-of-the-art Multicore Platforms. *Journal of Parallel and Distributed Computing*, 69(9):762–777, September 2009.

[64] PAPI: Performance Application Programming Interface, http://icl.cs.utk.edu/papi.

[65] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/parboil.php.

[66] K. Patel, E. Macii, L. Benini, and M. Poncino. Reducing Cache Misses by Application-specific Re-configurable Indexing. In *International Conference on Computer Aided Design (ICCAD)*, pages 125–130, November 2004.

[67] Linux Performance Counter Kernel API, http://user.it.uu.se/ mikpe/linux/perfctr.

[68] Ranger User's Guide, http://www.tacc.utexas.edu/ services/user-guides/ranger.

[69] B. R. Rau. Pseudo-randomly Interleaved Memory. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 74–83, May 1991.

[70] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, pages 134–142, 1990.

[71] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the International Symposium on Microarchitecture*, pages 187–198, December 2010.

[72] A. Seznec. A Case for Two-way Skewed-associative Caches. In *Proceedings of the International Symposium on Computer Architecture*, pages 169–178, May 1993.

[73] A. Seznec and J. Lenfant. Odd Memory Systems May be Quite Interesting. In *Proceedings of the International Symposium on Computer Architecture*, pages 341–350, May 1993.

[74] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[75] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power, and Area Model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[76] T. Sun and Q. Yang. A Comparative Analysis of Cache Designs for Vector Processing. *IEEE Transactions on Computers*, 48(3):331–344, March 1999.

[77] M. Teng. Comments on "The Prime Memory Systems for Array Access". *IEEE Transactions on Computers*, 100(11):1072–1072, November 1983.

[78] A. E. Turner. On Replay and Hazards in Graphics Processor Units. *UBC Masters thesis*, `https://circle.ubc.ca/handle/2429/43493`, June 2012.

[79] M. Valero, T. Lang, and E. Ayguadé. Conflict-free Access of Vectors with Power-of-two Strides. In *International Conference on High Performance Networking and Computing (Supercomputing)*, pages 149–156, November 1992.

[80] H. S. Warren. Hacker's Delight, 2nd edition.

[81] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[82] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. *Parallel Computing*, 35(3):178–194, March 2009.

[83] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

[84] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 235–246, March 2010.

[85] Q. Yang and L. W. Yang. A Novel Cache Design for Vector Processing. In *Proceedings of the International Symposium on Computer Architecture*, pages 362–371, May 1992.

[86] C. Zhang, X. Zhang, and Y. Yan. Two Fast and High-associativity Cache Schemes. *IEEE Micro*, 17(5):40–49, Sept./Oct. 1997.