PERFORMANCE BOTTLENECKS ON LARGE-SCALE SHARED-MEMORY MULTIPROCESSORS

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

Robert C. Kunz
December 2004

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

Dr. John Hennessy
(Principal Advisor)

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

Dr. Mark Horowitz

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

Dr. Christos Kozyrakis

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

Dr. Mark Heinrich

Approved for the University Committee on Graduate Studies:

Abstract

While multiprocessors have existed for many years, most parallel architectures are difficult to program efficiently. The key challenge is how to simplify the programming model so that programmers can write portable highly efficient parallel programs with minimal effort. For example, cache-coherent shared-memory architectures trade the memory system complexity of the coherence protocol for a simpler programming model that does not require communication to be programmed explicitly.

Software developers, operating systems researchers, and hardware architects have worked in their respective areas to improve the parallel efficiency of general-purpose applications. A critical question remains unanswered: how do advancements in each of these areas behave collectively? Using the FLASH machine, a large-scale cc-NUMA multiprocessor, this dissertation explores the interaction between hardware and software design trade-offs and quantifies the performance gains of memory system enhancements.

Researchers working on multiprocessor memory systems have advocated easing the programming burden by adding enhancements to the memory system designed to reduce memory latency and coherence overhead. Of course, these enhancements also add complexity and affect the speed of basic memory operations. Analogous to the lessons learned during the RISC movement over 20 years ago, this dissertation demonstrates that at large processor counts, simpler memory system designs are faster than more complicated ones, primarily because the additional contention present in the memory system overwhelms minor reductions in latency that more complicated protocols provide. Thus, architects should focus on minimizing memory controller occupancy on large-scale multiprocessors rather than just latency.

Even setting aside contention, the coherence protocol is a smaller bottleneck than other system aspects including the operating system's scheduling policies and the application's

effective or ineffective use of the cache coherent memory system. Our results indicate that to achieve reasonable performance, programmers still need to tune their programs to a specific architecture; such tuning limits portability. While coherence protocols might be able to provide a reduction in remote communication, the mismatch between an application and the architecture are often more significant and prevent major performance improvements.

Our results indicate that large-scale multiprocessors continue to remain difficult to program because the memory system alone cannot eliminate the need for programmers to remain aware of implicit communication. The software libraries, compiler, and operating system must apply complex machine-specific optimizations to reduce second- and third-order performance bottlenecks. Therefore, the memory system should provide meaningful visibility and feedback to programming monitoring tools and compilers. Without such tools to assist programmers, the programming advantages of a coherent shared memory multiprocessor versus a message passing multiprocessor are likely to be small for larger processor counts.

Acknowledgments

Do or do not. There is no try. –Yoda

Finishing the FLASH project has been the most challenging and yet rewarding experience of my professional life. Completing this dissertation would not have been possible without the efforts of many others who helped support me along the way.

First, I would like to thank John Hennessy, my principle advisor, for his guidance and mentoring over the last seven years. Working with him has been a dream of mine ever since my first computer architecture coarse over a decade ago. Despite his many commitments outside of the electrical engineering department, John has alway been there for me when I needed him. His efforts improved my research and writing tremendously.

I would also like to thank Mark Horowitz for serving as my secondary advisor. Through the years, he always looked out for his adopted students by setting the bar high. His feedback and encouragement improved my work tremendously. Mark Heinrich proved a valuable resource, even at a considerable physical distance from Stanford. I owe him a great debt for all of his time revising papers, discussing new directions, providing encouragement and some much needed humor. In addition, I thank Christos Kozyrakis for serving on my dissertation reading committee and Bernd Girod for chairing my oral defense committee.

My research would not have been possible without the foundation layed by the original FLASH team including Jeff Kuskin, David Ofelt, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Anoop Gupta, Mendel Rosenblum, and everyone else involved in the design and verification of the FLASH machine.

I would especially like to thank fellow members of John Hennessy's research group including David Ofelt, Joel Baxter, and Jeff Gibson for taking me under their wing and continuing to provide guidance even after they moved on to exciting new projects. I deeply respect their efforts and contributions to the FLASH project and my own work. Their

efforts taught me the skills to succeed.

In addition, I would also like to acknowledge the generous contributions of Kelly Shaw, Elizabeth Seamans and Ayodele Thomas during our weekly meetings over the last three years. Our research sessions helped refine my work and enabled us to exchange our ideas. Through their efforts, I learned about multiprocessors in other design spaces such as single-chip multiprocessors, network routers, and reconfigurable computers.

Mark Horowitz's research group provided a valuable resource as my research progressed as a forum for both testing ideas and remaining current in diverse research areas such as graphics and circuit design. Thank you to everyone in Mark's group including Bennett Wilburn, David Lee and my office mates, Francois Labonte and Jim Weaver.

Colleagues and friends are critical for success in the Ph.D. program. I especially would like to thank Dean Liu for his support and encouragement over the years. During our first years in the program, we were inseperable in classes and our friendship continues to this day. Robert Bosch gave countless hours of his time helping me as I learned the SimOS simulator. He proved to be not only a key resource but also a good friend.

In addition, Kinshuk Govil, and Ravi Soundararajan provided critical FLASH training and taught me humility through fantasy football leagues and Texas Hold'Em tournaments. Kinshuk taught me all that I know about operating systems and Ravi provided the initial remote access cache implementation.

I would also like to acknowledge SGI for providing the source code for the `IRIX6.5` kern and `libmp` OpenMP library. Without their generous contribution to our project, none of this analysis would be possible. I would also like to thank the Department of Energy for their gracious research funding for our project under grand number DE-FG02-03ER25564.

My extended family stood by me through my graduate program and I thank them for their love and support over the years. I especially thank my grandparents, Fredda (now deceased), Warren, Bernice, and Louis, my parents, Ron and Fran, and my sisters, Amy and Katharine.

Finally, I would like to thank my lovely wife, Meredith, for supporting me over the years. Her patience and love have been a godsend. She ensured that I took my research seriously enough to finish but that I wouldn't take myself too seriously and miss the journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel computing is a natural and old idea. The famous Illiac IV [9], developed in the late 1960s at the University of Illinois, is one of the earliest large parallel computers. This ambitious project ultimately took too long to design, cost too much to build and failed to deliver expected performance gains. In the intervening 40 years, large multiprocessors have continued to serve a specialized market for high-performance scientific or commercial applications for similar reasons.

In contrast, microprocessors have benefiting from dramatic advances in semiconductor technology and processor architecture, doubling in performance every one-and-a-half to two years. As a result, most people still use uniprocessor machines, based on the microprocessor. Pessimists have long predicted that Moore's Law will end sooner rather than later. Often, their dire predictions prove false because of the ingenuity of researchers and engineers who continue to develop novel fabrication techniques and processor architectures to improve microprocessor performance. In many ways, microprocessor performance growth has limited wider research and development of the general-purpose multiprocessor because there was no compelling performance reason to shift.

Lately, however, power constraints, longer on-chip wire delays, and limitations in instruction-level parallelism (ILP) are starting to limit microprocessor performance growth. Eventually fundamental physical communication limits and increasing power demands will limit our ability to improve microprocessor performance unless some fundamental paradigm shift occurs.

Parallel computing is one such paradigm shift. Once the microprocessor approaches

fundamental scaling limits, architects will leverage multiprocessors in an unprecedented way to continue performance trends. Already companies like Intel and AMD propose small-scale multiprocessor cores that leverage thread-level parallelism. Due to the programmer's heavy reliance on legacy software, the multiprocessor-programming model must remain as close to the uniprocessor-programming model as possible. In addition, programmers prefer efficient, yet *machine-independent*, program performance because the underlying multiprocessor architectures dramatically change as technology and machine sizes shift.

While small-scale cache-coherent multiprocessors are successful, many larger parallel machines abandon cache coherency. Somehow, the cache-coherent abstraction has failed to provide efficient, machine-independent performance with little programming effort at larger processor counts. The research and methodology presented in this dissertation seek to understand why programming large-scale multiprocessors remains difficult.

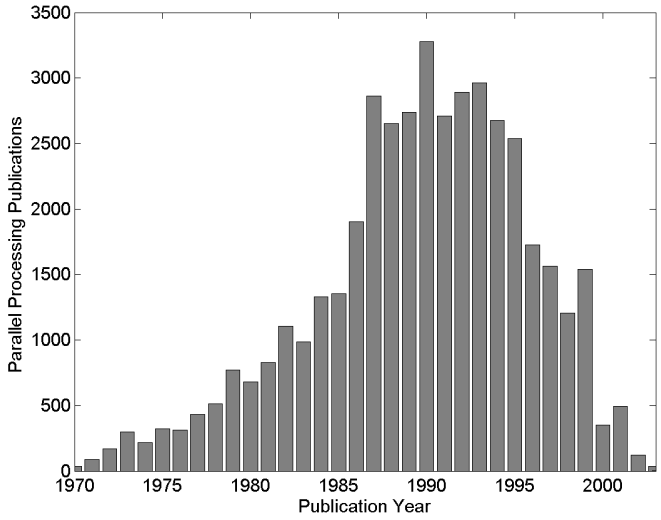


Figure 1.1: Multiprocessor and Distributed Computing Papers published between 1970 and 2004 [44]

The scarcity of useful large-scale shared-memory machines is not from a lack of effort by the architecture community. Figure 1.1 illustrates the rate of publications on multiprocessor and distributed computing papers from 1970 to today. Through the mid-1990s,

interest in parallel computing steadily increased because this area looked increasingly attractive. However, in the late 1990s, the community hit a research wall. Scarcer government funding, industry difficulties encountered in building and selling larger multiprocessor systems and a lack of new ideas from the architecture community contributed to the decline in parallel research.

Nor is the lack of these machines caused by the absence of parallelism in high-level applications. Parallelism frequently exists in these applications, but exposing this parallelism in software or extracting parallelism automatically in hardware proves challenging. The programmer, software interface, or parallel compiler fails to implement concise, machine-independent parallel algorithms. The hardware fails to provide a clear interface to the software or poorly extracts parallelism automatically.

1.1 Research Contributions

The primary contributions of this dissertation are:

- Measurements and analysis of performance and bottlenecks using a real large-scale multiprocessor to measure and analyze performance. The FLASH system [35, 36] was designed specifically to study design trade-offs present in large-scale multiprocessors. Using realistic multiprocessor provides a mechanism for studying a wider range of application complexity and data set sizes than what is typically available via simulation.

- Evaluation of operating system scheduling techniques to maximize system throughput when used in a high performance parallel processing context and a multi-user environment. This work illustrates that the traditional models of space-sharing and time-sharing fail to scale to larger processor counts, at least at a fine-grain level.

- Comprehensive analysis of complexity and performance trade-offs present in coherence protocol design for large-scale multiprocessors. This analysis consists of two parts: a practical study using coherence protocols implemented on the FLASH machine and a limit study using FLASH as an emulator to model ideal coherence protocols.

- Evaluation of software bottlenecks in high-level applications. These optimizations demonstrate that most of the performance bottlenecks present in high-level applications are well known problems with clear solutions in software—provided that the programmer is aware of the presence and location of bottlenecks in their programs and has significant understanding of the details of the architecture.

1.2 Organization of the Dissertation

To understand what types of problems limit the wider adoption of cache-coherency on large-scale multiprocessors, Chapter 2 explores scaling problems that led to the development of the shared-memory model and scaling problems encountered within high-level applications. The shared-memory model, developed in part to solve scaling issues encountered in earlier systems, and a variety of application-specific behaviors limit performance on these machines.

Naturally, we need to select a representative set of benchmarks that expose typical performance bottlenecks because often a trade-off exists between hardware and software solutions. Therefore, Chapter 3 discusses SpecOMP2001, a set of high-level applications that experience these application-specific effects, to provide a mechanism for exposing scaling problems. The next three chapters categorize why performance is lost at larger processor counts on the FLASH multiprocessor and explore potential solutions in hardware and software.

Unexpectedly, we encounter some bottlenecks related to the global scheduling of parallel programs and the virtualization of system resources provided by the operating system. Therefore, Chapter 4 investigates how the operating system influences—and in many cases degrades—performance at larger processor counts. Operating system-specific bottlenecks arise from multiprocessor scheduling policies that balance the needs of one parallel process against the throughput of the entire multiprocessor.

Perhaps we need only design a more efficient multiprocessor that exposes fewer pitfalls to the programmer. Chapter 5 explores how hardware architects have tried to do just this to reduce the challenge of efficient programming by increasing the memory system complexity. The FLASH multiprocessor was designed to provide an effective mechanism

for analyzing efficient and scalable memory system design for building larger multiprocessors. We find through this chapter that practical limitations restrict our ability to solve performance bottlenecks in hardware on larger multiprocessor systems.

Therefore, Chapter 6 shifts focus to the high-level application's software to evaluate the effectiveness of applying machine-specific tuning that remove performance bottlenecks. Using hardware-assisted instrumentation, the chapter identifies performance bottlenecks and removes them. The critical bottleneck remains the programmer's ability to apply well known—but difficult to identify—optimizations.

The dissertation concludes in Chapter 7 with a discussion of the fundamental barriers to effectively programming larger cache-coherent shared-memory multiprocessors.

Chapter 2

Evolving towards cc-NUMA Multiprocessors

The cache-coherent non-uniform memory access (cc-NUMA) multiprocessor originally emerged as an attractive architecture because the programming model presented to the user remains similar to the uniprocessor-programming paradigm. Surprisingly, while small-scale systems show some of the intended benefits of shared-memory, larger systems do not. The cc-NUMA architecture evolved precisely to provide a scalable and efficient programming interface. Furthermore, the expected performance bottlenecks are well known, but not all arise directly through the choice of the cc-NUMA architecture.

The FLASH machine, proposed, built, and evaluated over a period of 10 years, provides a valuable test-bed for analyzing the performance of high-level applications on cc-NUMA multiprocessors. However, during the last decade, multiprocessor architectures have shifted away from cc-NUMA and towards clusters of symmetric multiprocessors (SMPs) or clusters of standalone PCs. Traditional large-scale multiprocessors like the SGI Origin 2000 [38] proved too difficult to efficiently program high-level applications to justify the multiprocessor's large cost.

This chapter fleshes this story out in more detail to understand why multiprocessors remain difficult to program. Understanding this problem remains critical, because future single-chip multiprocessors are likely to follow a similar evolutionary track towards multiprocessor. Today, the number of processors per chip is small, but future chips could approach the number of processors present in typical SMPs.

First, the chapter reestablishes the key advantages and design goals of the cc-NUMA architecture. Section 2.1 defines key characteristics of the multiprocessor programming model. Section 2.2 explores how the cc-NUMA architecture evolved from the uniprocessor and earlier multiprocessor architectures and describes how each step changed the programming model. Every multiprocessor architect makes design decisions to partition key performance responsibilities among the programmer, compiler and software tools, memory system architecture and network. During each step of the evolution, key responsibilities shift between the software and the hardware.

Section 2.3 presents a systemic description of performance bottlenecks encountered when writing programs for cc-NUMA machines. The programmer and compiler hold most of the optimization responsibilities for these machines. Therefore, this section provides a guide to most of the pitfalls programmers are likely to encounter when writing for high-performance.

The differentiating feature of the FLASH machine is the programmable memory controller called MAGIC. In most other machines, the memory controller is hard-wired. While the latency overheads for replacing a hard-wired memory controller with programmable one are small, there are some FLASH-specific performance bottlenecks that are discussed in Section 2.4.

2.1 Programming Model Design

Programmers use a *programming model* interface to write correct and efficient programs for a specific architecture. Initially, programmers develop correct programs, and then apply tuning to increase overall performance.

Choices in architecture design often create a trade-off between ease of programming and overall hardware performance. By shifting the burden of program optimization away from the hardware, the programmer or compiler writer must manage the additional complexity of the programming model. However, this shift may simplify the hardware design or yield greater overall performance. The rise of the RISC processor typifies this relationship: simplifying the hardware allows the processor to benefit from faster clock speeds and instruction pipelining. However, the complexity of writing efficient programs increases

because the assembly writer or compiler arranges instructions correctly to express a program's semantics and hide pipeline stalls.

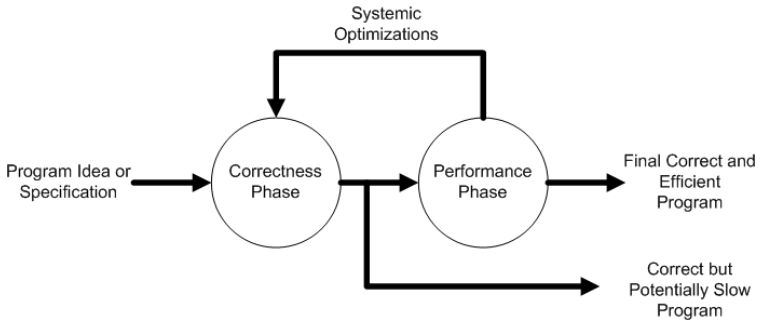


Figure 2.1: The programming design process

To simplify our understanding of writing programs, we divide the programming model into two design phases, illustrated in Figure 2.1. During the *correctness phase*, the programmer implements a new program idea, from a formal specification of a system, or an existing program written for another architecture using a programming language and the architecture's programming model. Once the programmer completes the correctness phase, the program precisely implements the original idea but potentially at suboptimal performance. Thereafter, the programmer enters the *performance phase* by applying optimizations that remove application-centric, architectural and application-created, and machine-dependent performance bottlenecks until the program achieves an acceptable level of performance. In the performance phase, the program's functionality does not change. The programmer may return to the correctness phase to remove high-level or systemic performance bottlenecks that require rewrites of major portions of the program.

The *memory consistency model* [20] portion of the programming model specifies the required relationship between load and store operations. Generally, programming under strong consistency is easier than under a weaker model because the programmer manages fewer memory ordering details. However, a strong consistency model requires the hardware to properly order memory operations.

The full complexity of performance phase depends on the final implementation details of the architecture. For example, a program running on a simple pipelined uniprocessor can leverage software pipelining to improve the performance of loops. However, correct

program execution does not require software pipelining. The optimal sequence of machine instructions depends on the instruction delays and functional units in the final hardware implementation of the processor.

An critical performance factor discussed in later chapters is the proper organization of a program's instruction and data memory in a distributed memory system. Proper *memory placement* is critical because some memory layouts can lead to poor cache locality and long memory latencies. Assuming a single address space, the program will execute correctly, if slowly, independent of how efficiently a program uses the caches or where data is physically stored.

2.2 Evolution of Multiprocessor Architectures

This section describes the evolution from the simple uniprocessor to the large-scale cc-NUMA multiprocessor and the resulting changes in the programming model. These changes shift the responsibilities of the programmer, compiler, and architect and impact the complexity of expressing functionality and performing optimizations.

2.2.1 The Uniprocessor Programming Model

The uniprocessor programming model is simple and straightforward: instructions complete in sequence. The simple and logical flow of operations leads to the development of programming tools that allow higher-level statement execution. Therefore, we can write a high-level language (e.g. `C`, `C++`, `FORTRAN`, `Perl`, `Python`, etc.) and develop compilers or translators to understand the low-level programming model. Dividing the labor between the programmer and the compiler simplifies programming in the correctness phase. The programmer only encounters the programming model as seen through the high-level language.

While the underlying hardware might transparently reorder memory operations to hide latency, it must maintain the uniprocessor's *strict consistency* model: any read returns the value of the most recent write.

During the performance phase, the programmer need only understand a few architectural details. Often programmers optimize uniprocessor programs by simply activating aggressive compiler optimizations to implement techniques like loop unrolling or software

pipelining. Capturing simple spatial and temporal cache locality by properly organizing data frequently happens automatically. Assuming the OS does not place pages in a pathological way (i.e. poor page-coloring algorithm), a program benefits from caching regardless of where the OS places data in memory. Some programmers may use tools to perform simple optimizations that reduce memory conflicts and improve performance.

The fewer programming model details required by the programmer during the correctness and performance phases, the easier the programmer finds writing correct and efficient programs. Multiprocessor architectures hope to keep the programming model equivalent to the uniprocessor—where both the correctness and the performance phases require little architecture specific knowledge.

2.2.2 Small-scale Symmetric Multiprocessors

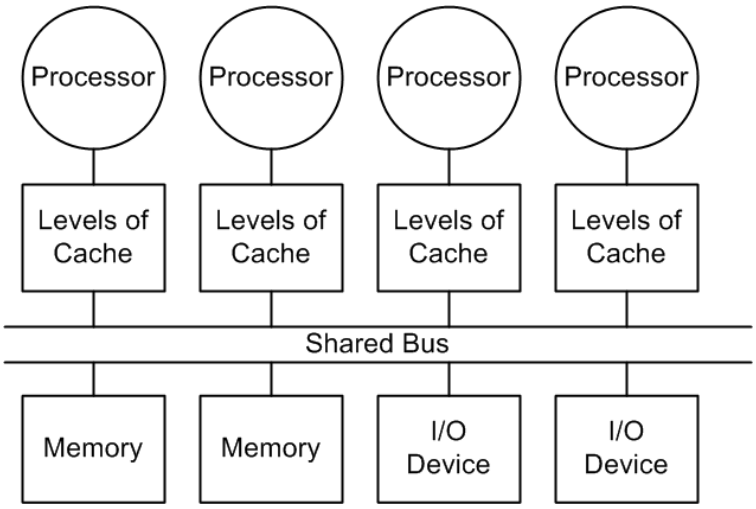


Figure 2.2: Symmetric multiprocessor architecture (SMP). Contention for the shared bus limits the effective size of this architecture

Uniprocessors naturally evolved into small-scale symmetric multiprocessors or *SMPs*. Additional processors and memories share information through a shared memory or a shared address space—in small-scale usually implemented with a coherent bus. Figure 2.2 illustrates a typical SMP configuration. The SMP, dubbed a *paracomputer* [53] in 1980,

predates more modern non-uniform memory access (*NUMA*) multiprocessors built during the early and mid-1990s. Modern SMPs include the Sun's Wildfire [14], Gemini Ultrasparc [31] and Niagara [34] processors.

The programmer has the additional responsibility of identifying parallel sections of the program and managing thread synchronization using a parallel program interface like `ANL macros` or `OpenMP`. This added interface increases the complexity of the correctness phase. Research projects such as the Stanford Hydra project [46] and parallel compilers such as Stanford's `SUIF` [4, 61] and `SAPIENT`, the University of Illinois at Urbana-Champaign's `POLARIS` [11] investigate ways of automatically discovering parallelism present in sequential programs. These tools simplify identifying and constructing parallel portions of the program but are not sophisticated enough to shield the programmer from understanding the parallel programming model.

To minimize the knowledge required by the programmer during the correctness phase, the hardware architect designs caches that snoop memory traffic broadcast across a shared bus to update their cache-line states. The *coherence protocol* specifies how to resolve data race conditions. This strategy keeps the correctness phase simple such that the programmer does not have to explicitly remove race conditions.

If multiple threads on each processor access the same memory location, memory operations broadcast on the shared bus provide *sequential consistency* [37]. Multiprocessor systems do not provide strict consistency because maintaining global timing proves too expensive. However, the sequential consistency model remains practically equivalent to the uniprocessor's strict consistency model. Thus, the programmer does not have to manage memory ordering during the correctness phase.

The performance phase is somewhat more complex than the uniprocessor's programming model. A program's memory accesses continue to benefit from temporal and spatial locality. An SMP's memory system provides a uniform memory access model (UMA); unloaded access time to each memory bank is identical. Therefore, the programmer does not have to manage memory placement beyond the effort required to tune uniprocessor programs. Communication among processes, however, leads to unavoidable cache misses, which are costly. The additional communication costs of shared cache misses are the key difference from the uniprocessor model.

Typically, SMPs use a broadcast model to communicate shared information between

processors. Every cache misses broadcasts across an interconnection network (typically a shared bus). In a system with few processors, network contention is usually negligible because the required bandwidth of the network to service cache misses is not high. However, as the number of processors increases both the length of the bus and the number of requests it must handle increase. Because the performance of a bus is inversely related to its length, eventually the required traffic exceeds the bus bandwidth, and some more complex interconnection network is needed.

2.2.3 Message-Passing Multiprocessor

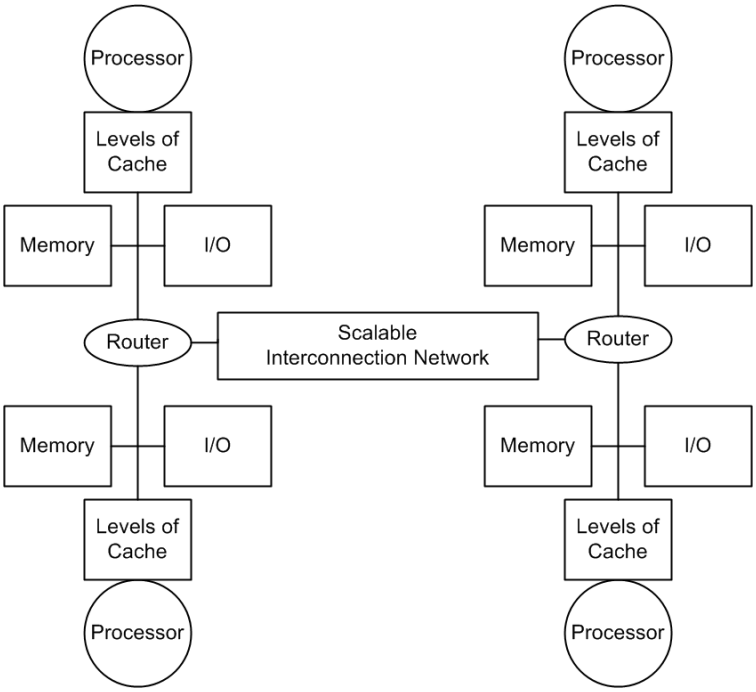


Figure 2.3: Message-passing multiprocessors scales well, but expressing communication proves difficult

To avoid contention present in broadcast models, architects turn to a distributed processor architecture where communication only occurs when explicitly requested by the processor. This architecture abandons cache coherence because broadcast operations are too expensive. The memory system no longer provides a shared address space between the

processors. Each processor holds only a portion of the total system memory. If a processor needs to access remote data, it must send a message to the remote node. Hence, this architecture is called *message passing* because every node must explicitly communicate with another node using a message. Figure 2.3 illustrates a typical message passing architecture configuration.

Because each processor contains independent address spaces, message passing machines are often referred to as *multicomputers* or clusters. For these machines, the hardware effort shifts to creating a scalable, high-speed interconnection network. To minimize communication overheads, smaller messages are often aggregated into larger ones. The memory system provides low-overhead send and receive protocols and data buffers to support arbitrary message lengths. A message passing multiprocessor places a large burden on the programmer, because in the correctness phase any communication or data sharing must be explicitly programmed.

Once the program executes correctly, the programmer finds tracking where communication occurs easier in message passing than in an implied communication architecture (i.e. an SMP) because the inter-processor communication patterns are well documented in the code. Once understood, however, fixing bottlenecks often is more difficult. In the performance phase, the programmer must repartition data among processors to reduce communication latency and network contention and this may require restructuring of the code.

The scale of modern message-passing multiprocessors demonstrates that these machines can scale to large sizes. Many message-passing systems are built with thousands of nodes, notably the Earth Simulator [18, 47] with over 5000 nodes, currently third on the list of top 500 supercomputers in the world [59]. Universities and research laboratories routinely connect workstations using a fast network to build larger clusters. Other examples include ASC-Q in Los Alamos National Laboratory with 512 nodes and 16-processor SMPs [6] and Lawrence Livermore Laboratory's Thunder system connecting 4096 Intel Itanium 2 processors running Linux [58].

Consider the University of Texas's `lonestar` 1024-processor supercomputer. Each node in the multiprocessor is a 3.06GHz Intel Xeon processor connected together using a Dell-Cray network. This system is number 40 on the top 500 supercomputer list. They claim an overall performance of 6.34 Teraflops at peak performance, which is approximately the theoretical peak performance of 6.12 GFLOPS for each CPU multiplied by

1024. The actual performance of the `LINPACK` [17] benchmark, used to compare all of the top 500 supercomputers, only achieves 65% of the theoretical maximum, or 4.15 Teraflops. FLOPS only measures how quickly these computers process math operations. The programmer is lucky to get a similar parallel efficiency from this machine for any general-purpose application without applying significant programmer effort.

The contract between the software, the compiler, and the hardware has fundamentally changed. The hardware design ceases to be the major bottleneck to scaling. The main obstacle to efficient performance becomes the programming model.

2.2.4 Simple NUMA Multiprocessors

An alternative approach, the NUMA multiprocessor, arose roughly at the same time as the message passing multiprocessors. Initially these systems were built without cache-coherence, because cache-coherence traffic on the shared bus was the barrier to scaling SMPs to arbitrary sizes. In such simple NUMA architectures, any processor can address and thereby access any of the memory distributed throughout the system. Non-shared remote loads and stores complete by accessing memory across the interconnection network and benefit from caching. Shared accesses maintain coherence by synchronizing at the memory banks, but shared data is never cached.

The correctness phase on this architecture is equivalent to the small-scale bus-based systems. The user does not have explicitly express communication or handle communication race conditions in shared portions of the program. However, the programmer must correctly identify all of a program's shared data segments.

The complexity of the performance phase requires more effort than SMPs because the memory system is effectively partitioned between shared and non-shared data. Because shared data remains uncached, the programmer must minimize sharing in a process similar to message-passing architectures in which communication occurs explicitly. Otherwise, the application's memory accesses do not benefit from caching at all. Experiences with the Cray T3D and T3E machines and DASH [39], a cache-coherent NUMA multiprocessor, suggest that effectively programming shared memory machines requires caching shared memory [28, 54]

2.2.5 cc-NUMA Multiprocessors

The natural extension of the simple-NUMA architecture is to allow the caching of shared data. This approach, called cache-coherence NUMA or cc-NUMA, allows the caching of data regardless of its sharing patterns through the use of a directory that holds cache-line state in memory (or in the cache line itself) and a coherence protocol. This architecture retains the simple shared-memory model present in the smaller-scale symmetric multiprocessors. Examples of this architecture include the Stanford DASH and FLASH machines, the SGI Origin 2000 and Origin 3800, the Sun Fire 15K, and the HP Superdome.

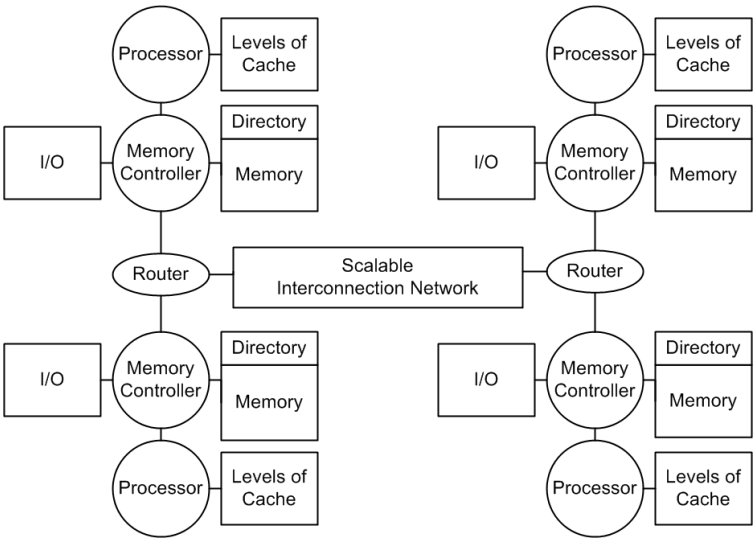


Figure 2.4: cc-NUMA Architecture. Design effort shifts to building a cache coherence protocol

Figure 2.4 illustrates a typical layout for a cc-NUMA machine. The memory controller arbitrates access to local memory between the local processor and remote requesters. The memory controller implements the coherence protocol to address communication races. Coherence operations are no longer broadcast across a shared bus, therefore cache interventions and invalidations—which on smaller systems would be handled by the caches snooping the bus and invalidating data automatically—are sent as coherence messages across the network.

The architect's responsibility is to design a memory controller that implements an efficient, scalable coherence protocol to service cache misses quickly. Most industrial multiprocessors have a hard-wired memory controller [38]. The FLASH multiprocessor uses an efficient custom-built protocol processor. Alternatives include the University of Wisconsin's Typhoon project [51], which replaced the memory controller with a commodity processor. The MIT Alewife [3] machine used a combination of custom logic and software traps to the processor.

Like the simple NUMA multiprocessor, the correctness phase of a cc-NUMA multiprocessor is similar to the SMP's correctness phase. Presumably in cc-NUMA architectures, the programmer manages fewer shared-memory details in the performance phase compared to simple NUMA. Shared and unshared data behave in identical ways and benefit from caching, therefore partitioning the memory space between shared and unshared is unnecessary. However, the true complexity of the cc-NUMA's performance phase remains unclear especially for larger processor counts. This dissertation explores exactly this question.

2.2.6 Summary

Figure 2.5 illustrates the relationship between multiprocessor architectures discussed in this section. The uniprocessor and message passing architectures form the ends of the functional and performance programming complexity spectrum. For some applications, architectures share similar performance complexities despite different programming models. While the programming models of simple NUMA and message passing differ dramatically, programmers must partition their data between shared and unshared portions for both. This effect is illustrated in the figure by overlapping boxes.

At large scale, can more aggressive coherence protocols that leverage remote caching deliver a simpler programming model? Alternatively, are architects forced to keep the memory system simpler for performance reasons? Understanding the trade-off between programming model and overall multiprocessor performance motivates the remainder of this dissertation.

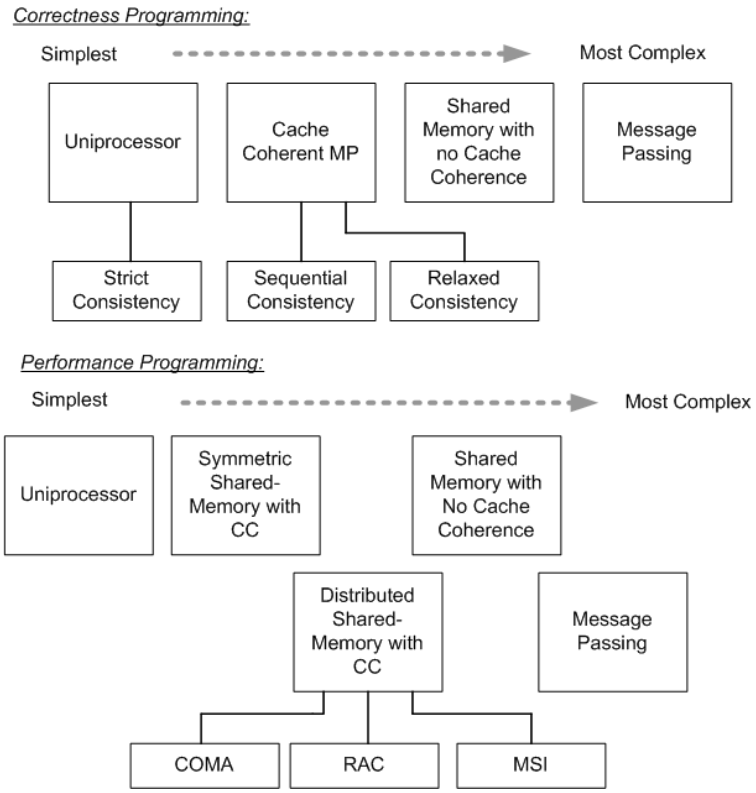


Figure 2.5: Complexity of functional and performance programming

2.3 Performance Bottlenecks Present in cc-NUMA Multiprocessors

The answer to the questions posed at the end of the previous section depends on which performance bottlenecks the programmer encounters when writing applications for cc-NUMA architectures. This section details most of the common pitfalls encountered when writing applications for shared memory. Performance bottlenecks fall into three general categories. Section 2.3.1 discusses architecture-independent bottlenecks created by the programmer when writing the application. Some bottlenecks form because of interactions between the application and the architecture. Section 2.3.2 presents those bottlenecks. Finally, bottlenecks arising due to particular machine parameters are discussed in Section 2.3.3.

2.3.1 Application-Centric Bottlenecks

Application-centric bottlenecks are bottlenecks the application would experience regardless of which architecture the application ran on, although, the precise impact of the bottleneck would vary from machine to machine due to processor speeds, memory sizes, and other machine-specific characteristics.

Removing or minimizing application-centric bottlenecks requires the programmer to apply high-level algorithm changes. Modern compilers apply peep-hole optimizations well, because the amount of information required is small. However, they are generally unable to make high-level algorithm changes because they lack enough global information to apply these complex parallel algorithm transformations.

Insufficient parallelism, Amdahl's Law for multiprocessors, is the most obvious example of this type of bottleneck. Most applications have some serial algorithm overhead or contain some code with too many dependencies to benefit from parallelism. By choosing and implementing a specific algorithm, the programmer implicitly limits the potential speedups present in the application. Clearly, data set size impacts the relative sizes of the serial and parallel sections of the application. Often increasing the data set size increases the size of the parallel section and thereby improves the available parallelism.

Improving the memory system's response time to memory requests reduces observed communication latency. Clearly communication affects speedup, but in the ideal the communication-to-computation ratio is programmer-determined because any parallel algorithm must communicate some information. The degree of ideal communication—the required minimum number of words to transmit between threads—determines the application-specific communication costs.

Less common application-centric bottlenecks arise from programming mistakes made in the parallel section that either cause excessive I/O traffic or OS system calls or traps. For example, on some operating systems each thread simultaneously calling `malloc` causes contention for kernel locks for virtual memory data.

2.3.2 Architecture and Application-Created Bottlenecks

This section presents bottlenecks created by interactions between the architecture and the application. These bottlenecks exist, to an extent, on all multiprocessors. However, the relative impacts of these bottlenecks depend on the cost of operations in the implementation.

Excessive Thread Synchronization

A parallel program may include multiple parallel threads that must coordinate with one another in a prescribed manner. However, race conditions between threads often create unintended and incorrect behavior. Thread synchronization mechanisms eliminate race conditions between parallel sections or access to shared data.

Frequent global barriers cause a problem if the cost of the barrier is high or if the execution times for parallel sections vary widely across threads. Any parallel thread must wait for all threads to arrive at the barrier. Therefore, all threads experience the worst-case execution time of any threads to the barrier. Most applications require barriers due to the presence of race-conditions between parallel sections, called *parallel-section dependencies*. If all the barriers are necessary for correctness, the programmer can only decrease number of barriers by changing the application.

Using a high-level parallel-programming interface like `OpenMP` often introduces implied barriers between every parallel section. The programmer can ignore these parallel-section dependencies. An intelligent compiler might be smart enough to identify and remove unnecessary barriers by checking for parallel-section dependences. However, the compiler has difficulty doing this type of high-level analysis because often the parallel sections reside in different procedures or are complex code segments in their own right.

The memory system could potentially resolve barrier waits as well by allowing execution of threads to continue beyond barriers and faulting or restarting in a stricter mode if a parallel-section dependence violation occurs [49]. More commonly, the memory system implements some hardware-assisted barrier mechanism such as fetch-and-op [23].

Unnecessary locking can also degrade parallel performance. Contention for locks causes parallel threads to execute serially. As the thread count increases, more serialization between threads occurs because the likelihood of lock contention increases. The memory system impacts this bottleneck because slow lock acquisition and release mechanisms could make the bottleneck worse.

However, hardware overheads are often trivial compared to the size of unnecessarily long critical sections. Clearly, the programmer could resolve the situation in software by shortening the critical section. An intelligent compiler might identify the smallest possible critical section required, using the programmer's lock acquisition and release code as a guide. The memory system could use a fault model similar to the barrier technique, which would work normally but this solution would only solve the problem if lock contention occurred infrequently.

Operating Systems Bottlenecks

The operating system virtualizes the hardware to provide all processes with an identical view of system resources. Most bottlenecks created by the OS design involve how virtualization overheads are exposed to the user.

Any uniprocessor OS schedules and deschedules threads to allow many processes share one processor. On a multiprocessor, the thread scheduler considers more variables when making decisions. As machine sizes scale and the underlying architecture shifts, the scheduler takes longer to decide. On a symmetric shared-memory multiprocessor, the scheduler's decision is easy because the cost of thread migration is small. However, on a cc-NUMA architecture, the scheduler must be more careful because the proper thread placement depends not only on the available processing elements, but also the size and location of a thread's data.

There are also bottlenecks associated with multiprocessor scheduling policies that allow multiple parallel programs to share portions of the machine in space and time. These bottlenecks increase an application's exposure to thread scheduling overheads and context switches if the multiprocessor scheduling policies create unnecessary thread interrupts. Also, scheduling policies can disrupt an application's data locality if threads migrate far away from their data. *Negative thread migration* significantly degrades memory system performance.

Communication-To-Computation Ratio Changes

The bottlenecks presented in this section are well known to multiprocessor architects. As an application scales to higher processor counts, the communication patterns for an application can shift. The cost of this type of application-centric bottleneck is determined by the use

of an architecture—a good example is a distributed address space (architecture visible) that affects the communication.

As an application runs on larger processor counts, the size of available cache increases. This cache aggregation effect causes the number of cache misses to decrease. Therefore, the performance of the parallel application improves significantly. While this effect is not a true bottleneck, cache aggregation can hide other bottlenecks by artificially inflating the observer performance. Thus, cache aggregation hides the true performance of the memory system from the programmer.

Alternatively, extra communication degrades the overall performance of the application. This communication arises from the overheads required for a specific algorithm choice. Most parallel sections introduce some per-thread communication overheads to set up the parallel algorithm or initialize temporary data.

A higher communication-to-computation ratio creates an effect similar to insufficient parallelism. The algorithm fails to scale because extra communication causes the parallel sections to run longer. Each parallel thread is doing more work at larger processor counts than at smaller processor counts. In most cases, the programmer must eliminate this extra communication by changing the algorithm or increasing the problem size. The compiler and the architecture can only mitigate the cost of remote communication because each cannot remove necessary communication.

Communication Hot Spots

Communication hot spots arise in parallel programs when frequently accessed data falls on the same node. Node-level false sharing is a special case of more general communication hot spots discussed later in this section. A small subset of the program's data can lead to contention on a single node.

Perhaps the program appropriately distributes pages among the parallel threads. Additional bottlenecks arise by the OS placing data on the page-level granularity. The local processor accesses most of a page, but a remote processor exclusively uses an unused portion. The remote requester must transmit across the network to retrieve the data, but it is the only requester for that portion of the page.

As processor counts increase, each thread in the parallel program accesses a smaller portion of the total memory required by the algorithm. Therefore, remote misses increase with processor count. This effect behaves in an identical fashion as increasing communication-to-computation ratio, but differs in that it is caused by page placement decisions made by the OS.

2.3.3 Machine Dependent Bottlenecks

Performance bottlenecks arise because of implementation choices made by the architects of a specific machine. In most cases, the programmer must be aware of these machine-specific problems when programming applications for performance. These coherence effects determine the difference between the ideal communication costs and the real observed memory latency. This section describes performance bottlenecks that are specific to the shared-memory model.

The memory system organizes memory into fixed-size hierarchical blocks to take advantage of temporal locality present in memory accesses and to reduce the complexity of memory management operations in software. Because data is aggregated into blocks, extra communication arises when multiple processors share resources in a block but do not actually unnecessarily exchange data. This *false-sharing* can occur at multiple levels in the memory hierarchy including the cache-line, a page, or even a node's memory.

Cache-line false sharing arises when multiple nodes exchange a cache line between them but access mutually disjoint sets of data words. Thus, this communication is unnecessary and expensive. This bottleneck is well known to architects and programmers alike. If cache line false sharing is systemic, an application will fail to scale well even at small processor counts. The number of nodes falsely sharing the cache line is bounded by the total words per cache line.

False sharing also occurs on a page or a node basis. Figure 2.6 illustrates three processors falsely sharing a cache line, a page, and node's memory. These bottlenecks share the same underlying property—processors falsely sharing a resource. However, each false-sharing bottleneck arises for different structural reasons in the program.

Page-level false sharing is less obvious to the programmer because the effect is often negligible on smaller machine sizes. The OS places data in memory on a page-level granularity. Page-level false sharing occurs when many nodes access the same page, but each

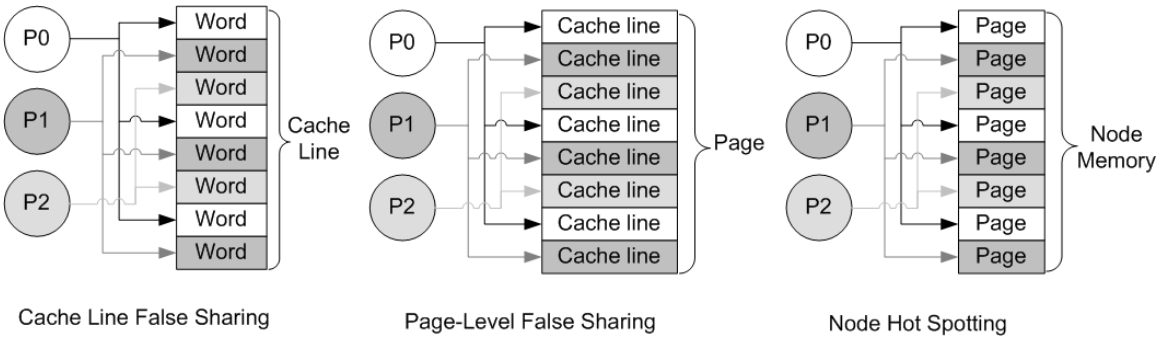


Figure 2.6: General false sharing types

node accesses disjoint sets of cache lines in the page. The total cache lines per page bound the number of nodes that could falsely share a page. The cost of page-level false sharing is bounded by the network latency to access the data on the page's home. However, at larger processor counts, contention arises as the accesses for the shared page contend with one another. This contention introduces queuing delay accessing the page's memory bank as processor counts scale.

Typically, page-level false sharing occurs on pages that hold thread-private data. The programmer solves the problem by separating unshared data into separate pages or segments. This process is similar to partitioning data between shared and unshared data in a simple NUMA machine. The potential requesters for a page scale with processor count. Therefore, on small-scale cc-NUMA machines, the programmer ignores this bottleneck because the potential requesters for a page remain small, contention is negligible, and applications benefit from the positive aspects of spatial locality.

Node-level false sharing occurs even if nodes access a disjoint set of pages on the same node. Contention occurs at larger processor counts where requests wait in the network buffer or the input request queues on the memory controller but access separate local memory banks. Frequently, this bottleneck arises from the master thread initializing and placing data before spawning parallel threads. This data placement causes all of the threads to access their data on the master node creating a large communication hot spot. This bottleneck arises from the programmer ignoring the distributed memory model and placing pages indiscriminately.

The compiler finds automatically removing false-sharing bottlenecks automatically difficult. The location of the bottlenecks in memory depends on the input set size, choice of parallel algorithm, and OS placement decisions. Either the programmer must find the bottlenecks and remove them by reorganizing data layout in software, or the memory system must adapt to migrate data closer to requesters. Two techniques discussed in Chapter 5, RAC and COMA, propose using remote request caching to mitigate these bottlenecks.

2.4 The FLASH Multiprocessor

Like any multiprocessor, the FLASH multiprocessor introduces machine-specific bottlenecks that impact performance. However, FLASH's architecture is unique in two important ways. First, the use of a flexible node controller provides a mechanism for identifying and quantifying performance bottlenecks. Second, the FLASH machine introduces machine-dependent bottlenecks that are not present in a typical cc-NUMA multiprocessor.

As mentioned earlier, many shared-memory multiprocessor architects propose removing bottlenecks presented in the previous section in hardware. The side effects of their proposals are primarily memory system complexity required to implement more aggressive techniques. Most researchers use simulation to evaluate the impact of these more complex proposals. Assessing the true costs and advantages of these proposals proves difficult for high-level applications with large data sets and long execution time because there are many opportunities for simulation error [22] and because simulation times are very long. Analytical modeling is not much easier because low-level implementation details must be accurate to produce meaningful results [56]. Fortunately, the FLASH multiprocessor provides a mechanism for exploring these designs in more detail using the same underlying hardware.

This study uses the FLASH multiprocessor [36] designed, built and evaluated over the last decade. FLASH is a 64-processor cc-NUMA machine with a programmable memory controller called MAGIC that runs software code sequences (*protocol handlers*) to implement the cache coherence protocol. The operating system is a modified version of SGI's `IRIX6.5`, which uses first-touch page placement. FLASH is binary-compatible with SGI's Origin 2000 [38]. Each node has a 225MHz R10k processor with 224 MB of

Table 2.1: Key Architecture Parameters for FLASH versus the SGI Origin 2000 and 3800

Paramter	*FLASH*	*SGI Origin 2000*	*SGI Origin 3800*
Year	1997	1997	2002
Processor Speed	225MHz	195MHz	400 MHz
Per-Node Main Memory	256MB	256MB	2GB
L2 Cache Size	2MB	2MB	8MB
L2 Cache Line Size	128 bytes	128 bytes	128 bytes
Local L2 Cache Miss	135 cycles	100 cycles	184 cycles
Network Topology	Double Mesh	Bristled Hypercube	Dual Fat-Tree

addressable main memory (32 MB are reserved for protocol data and directory memory overhead).

Table 2.1 summarizes key architecture parameters for the FLASH machines and compares them to the SGI Origin 2000 and 3800 [27]. A local L2 cache read miss takes 100 processor cycles on the 195MHz SGI Origin with a 100MHz HUB memory controller and 135 processor cycles on 225MHz FLASH with a 75MHz MAGIC. Remote latencies differ slightly because our network topology differs from Origin's hyper-cube. Point-to-point remote latencies are at most four times longer than local latencies.

FLASH's network topology differs from the original proposal in [35]. These differences arose because the original network was not deadlock-free. Appendix A describes the required network topology changes in more detail.

FLASH's network is fast enough that network congestion can be ignored as a performance bottleneck. Each link provides a bandwidth of 800MB/s. In order to saturate the network, a processor needs a sustained rate of 6.55 million remote cache misses per second or one L2 remote cache miss every 152ns or 36 processor cycles. Even in the best case, it is impossible for the R10k processor to generate this request rate because even local L2 cache misses that are serviced locally take 600ns. The R10k processor allows four outstanding requests. One local miss could be generated every 150ns. After four outstanding remote misses, the processor would stall decreasing the network bandwidth requirements. Perhaps a single link multiplexing requests from multiple nodes and virtual channels would approach the bandwidth limit for a brief period of time, but in practice this network congestion only occurs because of some higher-level behavior caused by coherence protocol traffic or a communication hot spot.

2.4.1 FLASH-specific Performance Bottlenecks

FLASH's programmable memory controller provides us with information from a real machine that normally is available only via simulation or analytic analysis. The use of MAGIC for instrumentation has only a minor impact on overall memory latency [21].

The critical difference between MAGIC and a hard-wired memory controller is the use of embedded handlers executing on the protocol processor that implement the coherence protocols on FLASH. These handlers benefit from on-chip direct-mapped 16KB instruction cache and a 1MB direct-mapped off-chip data cache that reduce the protocol processor's memory access time. The flexible processor introduces two types of additional overheads: longer access times and caching behavior of the coherence protocol's handlers.

The data caching behavior of handlers does not significantly impact performance because both a hard-wired and an embedded processor implementation benefit equally from data caching. Each handler operates on a small amount data, typically the 64-bit address, the message header, and the directory entry. The stack is small, a few kilobytes is sufficient because most handlers have short call depths. The data cache does not have to be large to be effective because if frequent cache reuse occurs, as in false-sharing for example, other bottlenecks degrade performance. More likely, data cache reuse is infrequent because memory accesses are randomly distributed.

However, the performance of coherence protocols on MAGIC depends greatly on the number of instruction cache misses, which represent extra delay that is absent in a hard-wired solution. MAGIC can capture some concurrency present in a hard-wired solution of handlers. The 2-way VLIW processor pipelines instructions and additional hardware performs pre- and post-processing operations of a handler simultaneously. Nevertheless, protocol handlers must be kept short to avoid conflict misses. The direct-mapped 16KB instruction cache is too small to implement arbitrarily long handlers.

2.5 Summary

Future machines will encounter identical scaling trends as the traditional multiprocessor. Single-chip multiprocessors are in fact at the beginning using small clusters of symmetric multiprocessors to build larger systems. Eventually, single-chip cc-NUMA machines will encounter similar scaling issues. This chapter details how the cc-NUMA model developed

to make functional programming easy. An open issue remains understanding how difficult are cc-NUMA machines are to performance program. While the bottlenecks are understood, the remainder of the dissertation uses high-level applications written for shared-memory to evaluate which of these performance bottlenecks proves the largest stumbling block to efficient performance at larger processor counts.

Chapter 3

Parallel Benchmarks

To this point, we have not defined what we mean by a ”high-level application”. Ultimately, selecting a benchmark set influences the results and central conclusions. This chapter describes why we focus attention on the SpecOMP2001 [62] benchmark suite, rather than the SPLASH-2 [62] benchmarks traditionally used in FLASH research, to represent high-level applications in later chapters.

Most users of larger multiprocessors are not primarily computer scientists or experienced software programmers. Instead, they are physicists, chemists, engineers or biologists who develop scientific applications to model systems relevant to their primary field of interest. They may have only a vague understanding that inter-node communication must be managed—the shared-memory model hides the explicit details of inter-node communication to keep functional programming simple. Computer architects should not require that users have earned an advanced degree in parallel architecture to write high-performance programs.

The multiprocessor programmer lacks a capable agent to apply machine-specific optimizations for them automatically. On a uniprocessor, a programmer relies heavily on the compiler to apply appropriate processor-specific optimizations. Even programmers who have detailed knowledge of a uniprocessor’s architecture rarely write more efficient code in assembly language. A novice uniprocessor programmer can cede performance programming to the compiler and achieve most of the potential performance of hand-tuned assembly. Parallel compilers have failed to succeed in an equivalent way at larger processor counts.

However, choosing a completely untuned benchmark suite is dangerous because a multiprocessor that is completely tolerant of programmer error would be impossible with these applications. No such machine would scale well. Thus we propose a middle ground in which the benchmarks have been optimized for the general architecture (e.g. shared-memory) without machine-specific optimizations. Some benchmarks will not be considered because they are either too optimized to the FLASH architecture or not optimized enough to the shared-memory model.

Multiprocessor researchers frequently use the SPLASH-2 benchmarks to represent average applications. Section 3.1 demonstrates that while the benchmarks have interesting remote memory behaviors, they have been tuned specifically with the FLASH machine in mind. Because they are primarily small kernels, they are too short and simple to accurately represent real applications.

As an alternative, Section 3.2 examines the SpecOMP2001 benchmarks, which use the OpenMP [5] application program interface (API) to express parallelism in a machine-independent fashion. The OpenMP API provides a standard mechanism for constructing parallel programs from a uniprocessor code-base. The interface and support libraries provide much of the architecture-specific tuning.

Since the SpecOMP2001 benchmarks are not as familiar as SPLASH-2, this chapter concludes with a high-level analysis of the SpecOMP2001 benchmarks using information typically available in a large-scale multiprocessor. This analysis illustrates the difficulty of understanding complex application behavior using only basic tools such as performance counters. However, we detect some common and well-known application effects, such as changes in communication-to-computation ratios or cache aggregation.

3.1 SPLASH-2 Benchmark Suite

During FLASH's design phase, the SPLASH-2 benchmarks were invaluable for testing and analyzing the architecture. The varied sharing patterns of the benchmarks allowed the designers to verify the cache coherence protocol processor design and develop new coherence protocols before the real hardware was available. The benchmarks had to be kept short because simulating the architecture was slow.

Table 3.1: SPLASH-2 Size and Time Characteristics

Benchmark	*Code Size (KB)*	*Data Size (MB)*	*Data Size (Nodes)*	*1p Time(s)*
Simulated Machine				
FFT	26	51	1	4
Radix	29	5	1	3
Ocean	155	124	1	15
Radix	22	16	1	3
Average	40	27	1	5
Real Hardware				
FFT	26	813	4	103
Radix	29	162	1	518
Ocean	155	1934	9	644
Radix	22	2155	10	508
Average	40	861	4	363

To generate short but interesting programs, larger and more complicated parallel application were reduced to simpler kernels to expose the memory system to intense remote communication patterns within a few seconds. As a result, the benchmark behaviors could easily be predicted and understood because they were small kernels executing with small data sets. The simulator provided a clear view into all parts of the MAGIC processor, which made analyzing bottlenecks easier. In some cases, some remote communication was unnecessary. Over time, many critical bottlenecks were identified and removed.

Once the FLASH real hardware was built, we were no longer constrained by simulation time and ran the benchmarks with larger data sets. Table 3.1 presents some basic characteristics of the most often used benchmarks in the SPLASH-2 suite for FLASH research. The first 5 rows illustrate parameters used when running the benchmarks on the simulated FLASH machine, and the last five illustrate parameters used to generate performance results on the real machine. The key differences are the data set sizes and the execution times. Most uniprocessor runs of the benchmarks complete within 10 minutes.

Using the real hardware, we eliminated some performance bottlenecks that were undiscovered in simulation. Typical multiprocessors have opaque memory systems that are difficult to quantify and analyze. J. Gibson's dissertation [21] demonstrated that coupling real hardware with FLASHPoint, a tool for correlating cache misses with specific data types and

code regions, facilitated machine-specific optimizations that dramatically improved performance. This tool reduced the opacity of the FLASH memory system with little overhead and recovered some information that was lost when transitioning from a simulator to a real machine.

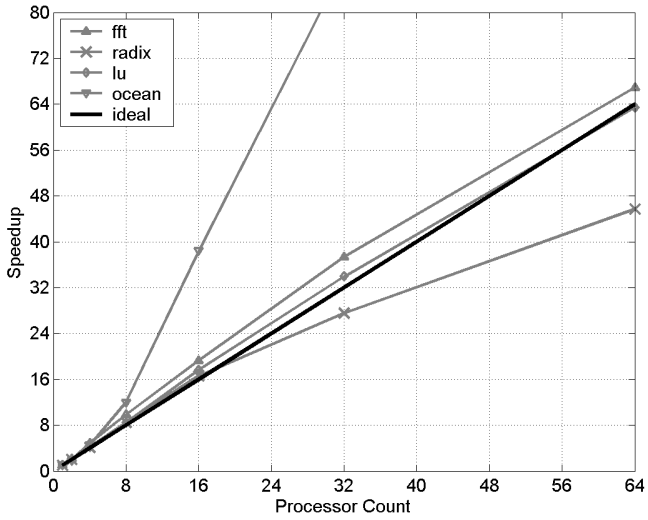


Figure 3.1: Speedups for the SPLASH-2 benchmarks

As expected, these benchmarks run efficiently on the real-hardware FLASH machine. Figure 3.1 illustrates the speedup curves for a subset of the SPLASH-2 benchmarks. `FFT` and `LU` achieve near ideal parallel efficiency at 64 processors. `Radix-sort` only achieves a parallel efficiency of 75% at 64 processors, due to load-balance issues. `Ocean` speeds up super-linearly due to cache aggregation. At 64 processors, Ocean speeds up by over 128.

The SPLASH-2 benchmark suite is not a good fit for further study. While all of these benchmarks perform extremely well, the degree of optimization applied to achieve these speedups is beyond what we expect a typical user will apply.

3.2 SpecOMP2001 Benchmark Suite

SPEC proposed the SpecOMP2001 benchmark medium suite in July 2001 as a series of benchmarks for analyzing and comparing performance of shared-memory machines using the OpenMP API. V. Aslot and R. Eigenmann developed the SpecOMP2001 [8] benchmark suite from the SpecCPU2000 benchmarks. In cases like `MGRID`, they used `POLARIS` [11], a parallelizing compiler, to extract parallelism from the uniprocessor version of the benchmarks. They tuned the applications using an UltraSparc II with up to four processors. Not surprisingly, V. Aslot and R. Eigenmann found that the benchmarks perform efficiently on one to four processors on an UltraSparc II.

The OpenMP API, developed for quickly parallelizing programs, simplifies machine-independent parallel expression. The SpecOMP2001 benchmarks are of particular interest because they express parallelism at an abstract level. This section first gives some background on the OpenMP parallel programming interface and then discusses the SpecOMP2001 benchmark suite in more detail.

3.2.1 Parallel Programming using the OpenMP API

The OpenMP API provides compiler directives, library routines, and environment variables to express parallelism in C, C++, and Fortran programs. The key advantage of the OpenMP API is that an average user can quickly construct parallel programs with minimal effort. Other approaches proposed include `pthreads` [10] (sometimes called `POSIX`) and the `ANL Macros`, which both express parallelism using low-level function calls.

```
1: int i, j;
2: for(i=0;i<N;i++) {
3:   for(j=0;j<N;j++) {
4:     a[i,j] = f(b[i,j],c[i,j]);
5:    }
6: }
```

Figure 3.2: Simple loop example

Many parallel applications are similar to the simple code example presented in Figure 3.2. This simple loop operates on three 2-dimensional arrays `a`, `b`, and `c`. The loop body reads each element in `b` and `c` and calls the function `f` to determine the new value of `a`. There are no loop dependencies so loops iterations can execute in any order.

While the overheads are larger when using OpenMP API than using a lower-level interface, novice users avoid all of the complexity inherent in the `ANL Macros` or other lower-level APIs. In most cases, this fine-tuning is unnecessary because the overheads are small.

```
1: int i, j;
2: #pragma omp parallel for private(j)
3: for(i=0;i<N;i++) {
4:   for(j=0;j<N;j++) {
5:      a[i,j] = f(b[i,j],c[i,j]);
6:   }
7: }
```

Figure 3.3: Simple parallel loop using the OpenMP API

The compiler and OpenMP parallel library automatically implement the numerous details of parallelizing loops—simplifying parallel construction and reducing opportunities for programmer error. Figure 3.3 shows the same loop using the OpenMP API. All that was added to implement parallelism was one compiler pragma at line 2. The pragma tells the compiler to keep `j` private. Each thread will use j to execute the inner loop, but its value is independent of thread count. The loop executes correctly regardless of the processor count.

To illustrate what the OpenMP API saves the programmer from managing, Figure 3.4 shows the steps required to parallelize the loop using the `ANL Macros`, a lower-level API, in a style similar to the SPLASH-2 benchmarks. First, the programmer must explicitly create all of the parallel threads, `P`, by calling the appropriate `CREATE` macro. Then each new thread enters the `LoopStart` procedure at line 9 and first must acquire its unique thread number, `MyNum` (lines 11-13). This atomic read-modify-write operation requires a lock variable initialized at line 2 and a thread lock and unlock at lines 11 and 13 respectively. Finally at line 16 each thread starts the main loop body. The threads call a barrier at line

```
 1: int i;
 2: LOCK_INIT(idlock);
 3: for(i=0;i<P;i++) {
 4:   CREATE(LoopStart)
 5: }
 6: LoopStart();
 7: WAIT_FOR_END(P-1);
 8:
 9: LoopStart() {
10:    int MyNum, MyStart, MyEnd, j;
11:    LOCK(idlock);
12:    MyNum = id++;
13:    UNLOCK(idlock);
14:    MyStart = N*MyNum/P;
15:    MyEnd = N*(MyNum+1)/P;
16:    for(i=MyStart;i<MyEnd;i++) {
17:      for(j=0;j<N;j++) {
18:         a[i,j] = f(b[i,j],c[i,j]);
19:      }
20:    }
21:    BARRIER(P);
22: }
```

Figure 3.4: Simple parallel loop using ANL Macros

21 to ensure all of the threads have finished executing the loop body (lines 17-19) before continuing to other sections of the program.

When using `ANL Macros`, there are many minor details that must be precisely implemented for the loop to execute correctly. If there are many loops that each stride through the `a`, `b`, and `c` arrays along different axes, managing loop indices becomes a difficult task. Finding a bug can be difficult because there are many opportunities for error and the application behaves in unusual or undefined ways if any of the small steps are omitted.

The `ANL Macros` code would have to be extended even further to work properly for a general number of processors, `P`. If `N` is not evenly divisible by `P`, the loop body will only iterate over subsets of the arrays. The loop iteration calculations in line 14 and 15 do not consider remainders. All of the SPLASH-2 benchmarks must be executed with a

power-of-2 number of threads for this reason. All of the loops are power-of-2 in size and thus have no remainder.

The OpenMP implementation has potentially larger overheads than the `ANL macros` implementation. On the FLASH machine, SGI provided the `libmp` source code that represents the OpenMP library function calls. For the SpecOMP2001 benchmarks, the overheads for the OpenMP library calls are minimal.

3.2.2 High-Level SpecOMP2001 Characteristics

This section presents some basic characteristics of the SpecOMP2001 benchmark suite to examine out-of-the-box performance. As discussed in the previous chapter, architects attempt to simplify the performance programming model by introducing memory system complexity. Analyzing out-of-the-box performance yields insight into how these benchmarks behave with only functional programming applied.

An absolute out-of-the-box execution time or parallel efficiency ratio can taint our results if some hidden effect artificially inflates or degrades performance. Specifically, the communication patterns of a benchmark may change as it runs on larger processor counts. Cache aggregation and communication-to-computation ratio changes are two common effects that shift our expectations of parallel efficiency. Quantifying these effects removes ambiguity about absolute performance measurements.

Initially, the base OpenMP libraries were opaque, which complicated this type of analysis because parallel loops were not clearly identified. There was no feedback from the unaltered `libmp` library to identify hot, frequently executed parallel loops or quantify serial time. Therefore, we modified the `libmp` library to instrument parallel sections and isolate serial time. To verify that our instrumentation was correct, we used prior work by V. Aslot and R. Eigenmann [7] that isolated critical parallel sections in SpecOMP2001 on a 1 to 4 processor UltraSPARC.

The SpecOMP2001 benchmarks are more complex than the SPLASH-2 kernels. Table 3.2 lists key memory size and execution time parameters for the SpecOMP2001 benchmark suite. Most of the benchmarks are considerably longer in terms of code size than SPLASH-2. The data sets are more comparable with the larger runs of SPLASH-2. However, the execution times are much longer. On average, uniprocessor SpecOMP2001 benchmarks take about 8 hours to run. This means that even intermittent OS behavior occurring

Table 3.2: SpecOMP2001 Size and Time Characteristics

Benchmark	*Code Size (KB)*	*Data Size (MB)*	*Data Size (Nodes)*	*1p Time(s)*
ammp	153	147	1	46085
applu	13	1474	7	26469
apsi	411	1631	8	23085
art	54	233	2	33907
equake	37	403	2	12001
fma3d	1729	994	5	32391
gafort	73	2668	12	62433
galgel	997	433	2	16451
mgrid	59	435	2	36854
swim	12	1547	7	31651
wupwise	59	1443	7	38665
average	98	740	4	29919

every few minutes might impact the performance if the OS adversely perturbs thread or data placement.

The AMMP and GALGEL benchmarks do not meet the minimum standard for parallelism potential discussed at the beginning of this chapter. They only achieve 40% parallel efficiency at 4 processors because of excessive cache-to-cache transfer misses. Without changes to the source, these applications cannot run successfully on most cc-NUMA architectures that operate on cache blocks. The performance problems with AMMP have been documented in other work with FLASH [21], and in fact SPEC dropped both applications from the official "large" SpecOMP2001 benchmark set.

The SpecOMP2001 benchmark suite consists of applications with diverse levels of local and remote communication. Figure 3.5 graphs the ratio of local cache misses relative to the total cache misses. It is an important ratio since applications are often sensitive to latency. Fetching data from remote memory takes much longer than accessing local memory on a cc-NUMA architecture. On FLASH, uncontended remote memory latencies are two to four times longer than accesses to local memory. Benchmarks that have a higher percentage of local misses, like SWIM and WUPWISE, will in general be more scalable. More aggressive hardware could eliminate remote communication using data migration, and benchmarks like APSI with high remote miss ratios should show benefit from these techniques.

Uniprocessor runs have unusually high remote miss rates due to lack of capacity on

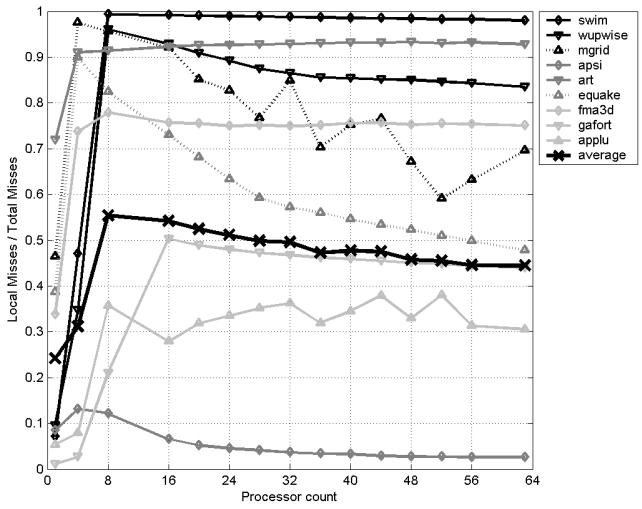


Figure 3.5: Local cache misses over total cache misses from 1 to 63 processors

the local node's memory. The memory footprints of the SpecOMP2001 benchmarks are larger than 224MB, the total memory size. As the benchmarks are run on larger machine sizes, *memory aggregation* creates super-linear speedups until all per-thread memory demands drop below the 224MB memory size. Only 25% of L2 cache misses are local on uniprocessor runs of the benchmark suite, but this ratio improves to 55% at 8 processors. Benchmarks like `SWIM` and `WUPWISE` that have high local miss ratios at larger processor counts have local miss ratios below 10% on the uniprocessor runs creating super-linear speedup until 8 processors. Using larger benchmarks would only increase the impact of memory aggregation.

The memory aggregation effect is specific to the FLASH multiprocessor and would not be a large effect on more modern systems. Most modern cc-NUMA machines have more available local memory. For most similarly sized benchmarks the memory aggregation effect would not exist, although larger applications could experience this effect. More importantly, the remote miss rate must drop as processor speed increases to maintain comparable efficiency. As an example, an SGI Origin 3800 system with 2 GB of memory for each processor has enough memory for all benchmarks except `GAFORT`.

To compensate for inflated parallel efficiency measurements that occur in moving from one processor to a small number of processors due to memory aggregation, the remainder of the dissertation presents parallel efficiency relative to the 8-processor results. Above 8 processors, first-touch placement succeeds for most data segments. Hence, a 64-processor run would achieve an ideal parallel efficiency of one if it speeds up by a factor of 8 over the 8-processor run.

While some scalability is lost due more remote communication present in larger systems, most of the benchmarks maintain constant levels of remote communication. The cache miss ratio indicates some broad characteristics about the benchmark suite. The `SWIM`, `WUPWISE`, `ART` and `FMA3D` benchmarks have high locality that remains constant from 8 to 63 processors. `APPLU` and `APSI` show the opposite trend with low local cache miss ratios of 25% and 3% respectively. `GAFORT` sits in the middle at 50%. `MGRID` and `EQUAKE` clearly experience a decrease in cache miss locality as processor counts scale above 8 processors. `MGRID`'s ratio is a bit more erratic with frequent drops and increases in locality but the general trend is clearly down. This change in memory locality comes from an increase in communication-to-computation ratio as processor counts increase.

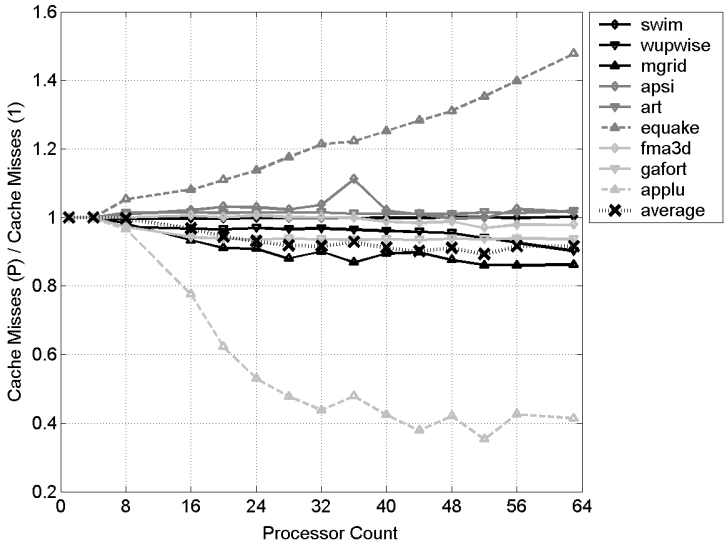


Figure 3.6: Total L2 cache misses for the SpecOMP2001 benchmarks from 1 to 63 processors

Most benchmarks maintain flat communication-to-computation ratios as processor counts scale higher. Figure 3.6 illustrates the total L2 cache miss counts from 1 to 63 processors for each benchmark. All counts are normalized to the number of cache misses on a uniprocessor. Most of the SpecOMP2001 benchmarks do not show significant changes in the total cache misses.

APPLU, an exception, experiences *cache aggregation* so its performance will inflate. As system sizes grow, the total available cache size increases. Eventually, large unshared data segments fit into the available caches, dramatically reducing the average memory latency. Therefore, this benchmark should naturally show *super-linear* speedup because of cache aggregation. If not, the benchmark fails to use all of its parallel resources efficiently.

Conversely, the total number of cache misses increase linearly with machine size in EQUAKE. The additional communication present at higher processor counts limits the maximum parallel efficiency. Without changing the parallel algorithm, we would be satisfied with EQUAKE's performance if it executes with 50% parallel efficiency.

Architects typically simplify communication-to-computation ratio analysis error by scaling the problem size up to ensure that 100% parallel efficiency is the ideal target for all benchmarks. Two pitfalls, irregular communication patterns and memory aggregation, must be avoided to accurately compensate for increasing communication. The first pitfall is minor: the communication changes do not follow smooth linear functions—especially in MGRID where the remote misses generally increase but rise or fall from one data point to the next. The physical per-node memory sizes are so small on FLASH that memory aggregation becomes more of an issue as the problem size increases.

Parallel efficiency analysis requires additional steps for APPLU, EQUAKE, and MGRID benchmarks in order to correctly understand parallel efficiencies. Later chapters of this dissertation include larger input sets to support claims about the parallel efficiency or lack thereof in these three benchmarks. Unless explicitly stated, the benchmarks results do not show the effect of scaling the problem size.

The SpecOMP2001 benchmarks generate a diverse range of request rates to the memory system. Benchmarks like SWIM with higher cache miss locality places higher demands on the memory system to service memory system requests quickly. Local cache miss latencies are shorter than remote cache miss latencies; applications with natural locality generate requests at a faster rate since the processor is stalled less often. Figure 3.7 shows the rate

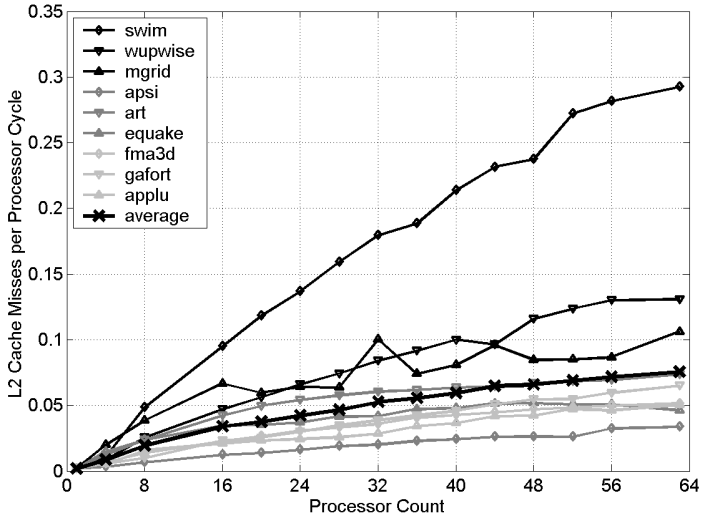


Figure 3.7: L2 cache misses per R10k processor cycles

of L2 cache misses per R10k processor cycle. The three applications with the highest local cache miss ratios, `SWIM`, `WUPWISE`, `ART`, and `MRID`, place the highest demand, measured by the cache misses per processor cycle, on the memory system. `APSI`, with only 3% local misses on average, places the lowest demands on the memory system.

3.3 Summary: SpecOMP2001 Out-of-the-Box Performance

Most of the SpecOMP2001 benchmarks meet the criteria established in the beginning of this chapter. There are some optimizations for shared memory, but no additional machine-specific optimizations like in the SPLASH-2 benchmark suite. Figure 3.8 presents the speedups of the benchmarks. Unfortunately, the out-of-the-box performance of the benchmarks at larger machine sizes is poor. This benchmark suite on average only scales with a 15% parallel efficiency from 8 to 63 processors.

The remaining chapters explore why these benchmarks scale poorly and detail which parts of the architecture contribute most to this performance loss. Chapter 4 discusses several operating system decisions that adversely affect performance. Once the operating

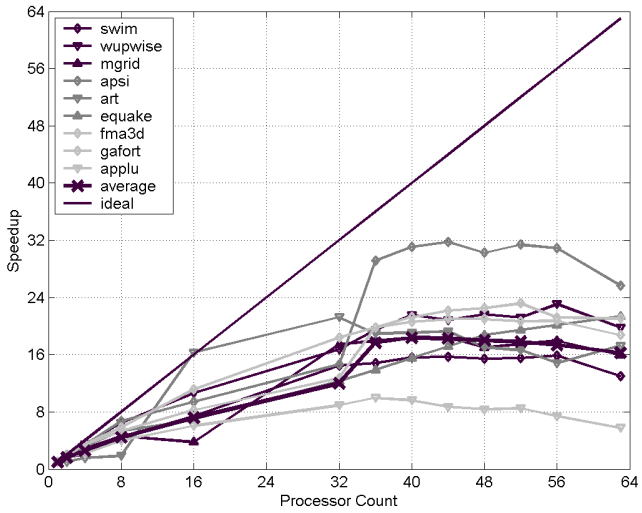


Figure 3.8: Out-of-the-Box speedups of the SpecOMP2001 benchmarks

system issues are addressed, the parallel efficiency improves to 63%. One might expect that the remaining performance is lost in the memory system. However, Chapter 5 challenges this assumption and demonstrates that the memory system itself is a small part of the total remaining performance loss. Chapter 6 presents some basic machine-specific software optimizations required to achieve near ideal parallel efficiency of 83%.

Chapter 4

Operating System Performance and Bottlenecks

Virtualization of the processor, memory system and other shared resources serve as a central pillar of the interface provided by the operating system. This interface allows programmers to write a single application without explicitly managing system-level details such as page placement, processor scheduling, or overall system throughput because the operating system arbitrates access to shared resources. As a result, uniprocessor applications are extremely portable. Similarly, programmers prefer a similar interface to system resources on a cc-NUMA multiprocessor. However, multiprocessor scheduling overheads arise when we move to a parallel execution model on a cc-NUMA multiprocessor. This chapter measures these overheads and observes that these operating system overheads are in fact quite high on larger sized machines. While the remedies to these bottlenecks are simple, they expose a critical flaw in the scaling of the operating system interface to larger machine sizes.

By definition, cc-NUMA machines distribute memory around the system. When a process allocates memory, the OS needs to decide in which node's memory to place the new data. On a uniprocessor, this decision is simple because there is only one memory. On a distributed system the decision is harder only if a processor's local memory is full.

First-touch page placement works well in most cases where the programmer has written a parallel application with memory locality in mind. When a processor touches an allocated, unplaced region of memory, the OS places the data on that processor's local memory. If the local memory is full, the OS places the data in another node topologically

close to the original processor. `IRIX6.5` provides additional system calls for implementing round-robin or fixed page placement policies. This chapter omits further discussion of data placement issues because most techniques are well understood. However, later chapters discuss data placement techniques in more detail since it is an important component in memory system and application tuning.

Significantly more difficult to develop on a multiprocessor OS are policies that achieve locality-sensitive scheduling and efficient system throughput. A uniprocessor has only one processor to schedule so a simple round-robin queue coupled with a priority scheme works well. When a process uses up its time quantum, the OS de-schedules the process and places it at the end of the queue. Thus the user perceives an illusion of many processes running simultaneously. On a multiprocessor, many processes can run simultaneously on different processors, so the OS's job of managing processes becomes a larger challenge.

Programs, processes, and threads are terms that are occasionally used interchangeably. This chapter formalizes the meaning of these terms to simplify further discussion. A programmer writes a parallel program. The *process* refers to the parent process initially created by the OS when a user executes a program binary and all of the child threads created by the parent process. A *thread* are child threads created by parent process that participate in the parallel execution of the program. The OS schedules threads and the programmer creates processes by executing program binaries. A process can use all the processors available, P, by creating P threads, whereas a thread only executes on one processor.

4.1 Multiprocessor Scheduling Policies

The operating system balances the need for maximum throughput of all threads against the individual processor and data requirements of a single thread. Threads share both space, meaning neighborhoods of processors, and time. The operating system's *job scheduler* make two decisions: when to schedule threads in time and where to place scheduled threads in space. Uniprocessor systems only require time-sharing policies because there is only one physical processor executing scheduled threads.

4.1.1 Time-Sharing Techniques

Time-sharing policies implemented in the job scheduler allow multiple threads to share processors through multiplexing control of the processor over time. Using time-sharing is an especially important technique when there are more threads available than there are processors, as in a uniprocessor. Related threads are likely to communicate or synchronize with each other, so scheduling related threads together provides benefits in overall execution time and system throughput.

The *batch scheduling* policy schedules parallel processes on a global parallel job queue. Operating systems provide batch scheduling using software like Network Queuing System (NQS [32]) that process jobs in a first-come, first-serve, first-fit order. The batch queue starts the first process on the queue that can execute given the current number of idle processors. If a process requests more processors than are available, the batch queue delays the process until currently running parallel processes complete. Processes wait as long as there are insufficient resources to execute the process. Wait time in the batch queue is potentially long. However, once a process finally executes on the multiprocessors, it has exclusive and dedicated access to system resources.

The default *run-queue scheduling* policy aggressively schedules threads on idle processors. If any processor is idle, the scheduler checks the local run queue on the idle processor. If that run queue is empty, the scheduler checks the global run queue for any pending threads. If the global queue is empty, the scheduler grabs threads off of another processor's local run queue. If a thread is descheduled, it is placed on the run queue of the processor it was executing on previously. This policy minimizes the wait time of all threads on queues. However, thread migration occurs under heavy system loads.

The *gang scheduling* [48] policy benefits a process by scheduling related threads concurrently. The scheduler treats a gang-scheduled process as a thread. All threads in a process are scheduled and descheduled together as a block. The process will only execute if the job scheduler can schedule all of a process's threads at the same time. Similarly, the job scheduler deschedules all parallel threads associated with a process if any individual thread deschedules. In this way, if two processes attempt to use the entire machine, they share resources much like two threads running on a uniprocessor. Each process uses the multiprocessor for a fixed quantum and then yields the multiprocessor to the competing process.

There are several known disadvantages with gang scheduling. Contention for centralized scheduling control can increase the cost of context switches. A thread may need to repopulate its cache. Finally, multiple gang-scheduled processes can cause process fragmentation [48]. Despite the disadvantages of gang scheduling, Burger et al. [12] concluded that gang scheduling was required for effective multi-user performance on distributed shared-memory machines.

The process must request the gang scheduling policy. The `libmp` library enables gang scheduling by default to ensure that OpenMP programs executing concurrently share resources fairly. The overhead is the context switching time associated with this scheduling mechanism. Rescheduling time can be significant since enough processors must be available to schedule all threads together since `IRIX6.5`'s scheduler weighs gang-scheduled threads and regular threads equally. SGI's `libmp` library dynamically throttles the thread count to run with as many available processors as possible. The throttling mechanism varies thread counts depending on the machine's load adding variation to parallel efficiency, so we disable it.

4.1.2 Space Scheduling Techniques

Space-scheduling policies distribute multiple threads among different processors of the machine to balance load. For example, two 4-processor jobs run concurrently on mutually exclusive sets of processors of an 8-processor machine without interfering with one another. Space-sharing techniques are useful when there are at least as many processors as threads.

`IRIX6.5`'s job scheduler distributes threads among local and global run queues. The OS moves unplaced threads to the global run queues. The scheduler must keep scheduling decisions short. The more time the scheduler takes to decide where to place threads, the less time the compute processor executes user programs.

The scheduler by default places threads using a *cache-affinity* policy. By default, the job scheduler places a thread on the run queue of the processor that last executed the thread. Thus a thread executes on the node that holds its cached data. Thread migration occurs when another idle processor grabs the thread from a local processor's run queue before the processor re-schedules the thread. This policy works well for compute-intensive applications that have small data footprints and a high degree of cache reuse [60].

The cache-affinity policy was started because moving data between caches is expensive. Each cache line must first be removed from the source cache using interventions and invalidations, which disrupt the processor's execution flow. The data must also be sent through the network to the destination cache. Using a cache-affinity policy reduces the need for cache migration and ensures that a thread avoids repopulating its cache.

If process migration occurs, eventually the data will move to the new processor's cache. However, if a data-intensive application experiences a high frequency of L2 cache misses to local memory, thread migration causes a more permanent increase in L2 cache miss times because cache misses now must transmit across the remote network. Under cache-affinity, once a thread has migrated to a new processor, the operating system will place the thread on the new processor's run queue, not the original processor's run queue, until another migration event occurs.

A *memory-affinity* policy places a thread on the processor where its memory has been placed by the operating system. When a thread is descheduled, the job scheduler places the thread on the run queue of the processor that holds the thread's memory. Migrations to other processors are allowed. Unlike the cache-affinity policy, the process migration is temporary except under heavy load. Once the migrated thread is de-scheduled, the job scheduler places the thread back on the processor that holds the thread's data.

Normally, the scheduler ignores information about a processor's memory demands. If a thread has moved among many processors or if the parent process placed most of the memory, the proper placement is ambiguous. `IRIX6.5` simplifies this system by ignoring memory-affinity unless the process explicitly sets up a *mld*, or memory locality domain, for each thread. A memory locality domain gives a simple hint to the scheduler about which subset of processors hold a thread's memory, called the *mldset*. If an mldset has been created, the schedule uses the hint instead of the default cache-affinity policy. SGI's `libmp` library sets up an mld for each thread based on the initial processor that executes the thread. The mldset is only a hint. Another processor outside of the mldset might execute a waiting thread if its local run queue and the global queue are both empty.

A memory-affinity policy can also be enforced in software by *thread pinning*. The pinned thread forces the job scheduler to always place the thread on a specific processor. Migration never occurs. Nodes topologically close to one another in the network naturally experience shorter remote memory latencies. Therefore, pinning a process's threads in a

common tile or partition effectively captures some of this remote memory access locality.

4.2 Unloaded Overheads of Scheduling Policies

This section presents the measured overheads of multiprocessor scheduling policies on an unloaded machine. This analysis assumes that the batch scheduling policy's execution time is equivalent to a system with pinned threads and gang scheduling disabled.

4.2.1 Methodology

To eliminate serial time from our analysis, we place cycle counters in the `libmp` library before each parallel section, measure the overall execution time including synchronization, and then subtract the time spent in parallel sections. This change represents a simplified version of the application because there may be times when more than one but less than a full complement of processors are executing. However, the SpecOMP2001 applications are parallelized to take advantage of all available processors, so the error is small.

Experiments in this section consider the absolute execution differences between these policies. The 8 processor runs differ depending on the applied time-sharing or space-sharing policy. The uniprocessor execution times are identical. Therefore, in this section parallel efficiency is measured from the same uniprocessor run for each application.

Disabling gang scheduling is trivial. SGI provides an environment variable for switching gang scheduling on and off. However, gang scheduling is on by default. So the out-of-the-box numbers presented in the previous section include gang-scheduling overheads.

Implementing the appropriate space-sharing technique requires more effort. Initially, the `libmp` library only created an mldset for each thread if the library detected that it was running on an SGI Origin machine. Unless debugging options are active, the user remains unaware that the library uses a cache-affinity policy. SGI provided Stanford with the source to the `libmp` library, so we modified the library to create mldsets on the FLASH machine to implement memory affinity scheduling. However, this remedy is atypical. Normally, the programmer uses compiled libraries, and remains oblivious to the underlying implementation.

Pinning threads requires modifying the SpecOMP2001 applications to explicitly call a special procedure, called `pin_threads`, to tell the operating system to explicitly place

```
1:  max_cpu = max_flash_nodes();
2: #pragma omp parallel for private (my_cpu)
3:  for(thread=0; thread<num_threads; thread++) {
4:      pin_cpu = max_cpu - thread - 1;
4:      sysmp(MP_MUSTRUN, pin_cpu);
5:      my_cpu = sysmp(MP_MUSTRUN);
6:      if (my_cpu!=pin_cpu)) {
7:         error;
8:      }
9:  }
```

Figure 4.1: Pinning threads code example

threads on a specific processor. Figure 4.1 presents a simple technique for pinning threads using the OpenMP API. This example places threads in a top down fashion. The master thread executes on the top processor.

Top-down thread pinning proved necessary because FLASH's boot processor, processor 0, performs slower than other processors. Idle and operating system requests naturally hot-spot at processor 0's memory controller. In addition, many OS data structures and kernel-specific pages reside on node 0, so there is less available local memory. Threads placed on node 0 experience greater contention for local memory and higher probability of first-touch placement failure.

When using the memory-affinity policy, the `IRIX6.5` scheduler likely will place a thread on the boot processor when executing a 63-processor program. This overhead can decrease performance by 10%-20% for reasons related to—but not caused by—the operating system's scheduling decisions. The scheduler could easily be modified to avoid processor 0 when scheduling new threads, but that might introduce unintended side effects—including causing unnecessary thread migration.

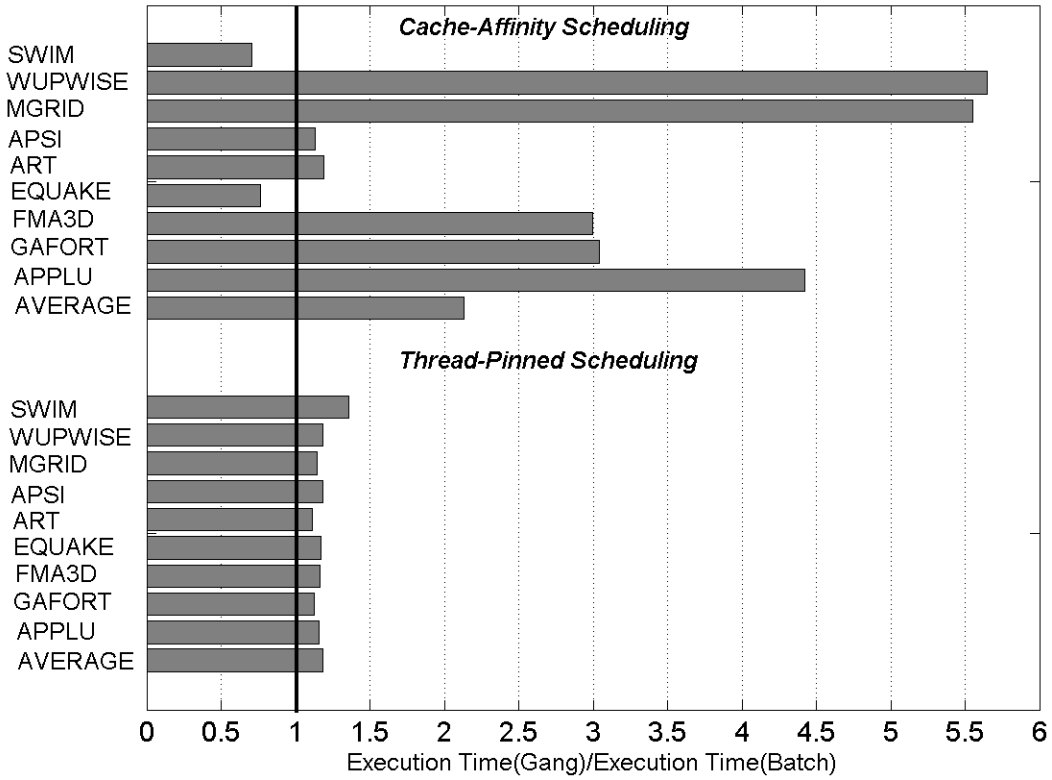


Figure 4.2: Gang scheduling overheads at 63 processors. Higher ratios represent larger overheads over batch scheduling

4.2.2 Results

We observe that scheduling decisions made by the operating system are a major factor in losing parallel efficiency. Figure 4.2 shows the impact of gang scheduling using two separate space-sharing policies on the SpecOMP2001 benchmarks. Each benchmark's gang-scheduled execution time is divided by the batch-scheduled execution time. The higher the ratio, the worse the gang-scheduling overhead relative to batch scheduling.

From a high-level perspective, gang scheduling introduces overhead as each benchmark is involuntarily interrupted frequently. Context switches—most involuntary—remain constant near 8 per thread per second. When gang scheduling is turned off, the rate of context

switches drops to 0.14 context switches per thread per second. In the latter case, most context switches are voluntary yields of the processor.

More surprising, the gang-scheduling overhead varies more when using a cache-affinity policy than explicitly pinning threads. Gang scheduling slows the benchmarks by 54% under the cache-affinity policy and by only 15% under a thread pinning policy.

Under the cache-affinity policy, threads migrate away from their data since physical memory location is not the primary factor in scheduling. This migration causes both a temporary performance loss as data must be brought into the cache again and a more permanent effect of forcing some formerly local data to become remote. The more optimized an application is for NUMA, the more migration decreases performance. Pinning threads increases parallel efficiency by 247% when threads are pinned to a specific CPU.

This migration process occurs in all time-sharing policies. However, the impact is greater under gang scheduling because of the higher frequency of context switches which increase the probably that the OS will schedule other unrelated processes.

Pinning threads to a specific CPU removes the negative thread migration effect. The only overheads that remain are the frequent context switch time and the individual thread's wait time for the competing thread to yield its processor. The gang-scheduling overheads all fall within 11% to 27%—essentially a constant overhead.

The execution time of `SWIM` and `EQUAKE` *improve* under the gang scheduling and cache-affinity policies, which should not occur given the high frequency of involuntary context switches. There are two potential causes for the anomalous benchmarks. Negative thread migration could occur in a pathological way that exposes these batch-scheduled applications to higher remote memory latency and migration overheads. Alternatively, frequently interrupting the application could reduce memory system contention and improve the wait time of memory system requests enough to negate the gang-scheduling overhead.

Figure 4.3 isolates the impact of space-sharing techniques on the benchmarks with gang-scheduling disabled. Thread pinning performs the best of all three policies. This result makes sense because explicitly pinning threads removes any possibility of negative thread migration. A memory affinity policy captures most of the benefit of pinning threads because thread migration becomes unlikely.

The default cache-affinity policy implemented by the `libmp` library severely degrades performance. The `libmp` library only implements a memory-affinity scheduling policy

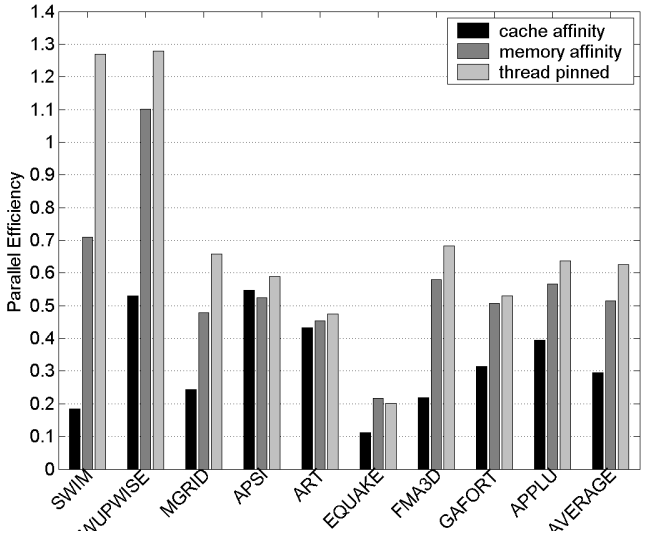


Figure 4.3: Parallel efficiencies for space scheduling policies at 63 processors

on SGI Origin machines. This oversight was quickly corrected once discovered, but the process illustrates the importance of writing machine-independent libraries and parallel code.

Applications that precisely control thread placement benefit from shorter remote memory latencies if they are more likely to share data with a few remote sharers, as in the `SWIM`, `WUPWISE`, `MGRID`, `FMA3D` and `APPLU` benchmarks. If remote requests are distributed more evenly around the network, applications still benefit from avoiding node 0. The memory affinity policy runs schedule a thread on node 0, which slows the entire program. `APSI`, `EQUAKE` and `ART` share similar execution times regardless of space-sharing policy.

`APSI`'s remote L2 cache misses are distributed evenly around the network, so thread migration only costs the cache transmission time. There is no memory locality for migration to destroy. The memory accesses behave more like a symmetric memory system where all of the cache misses have identical latencies.

The `EQUAKE` benchmark shows an unusual behavior: using the memory affinity policy performs *better* than the thread pinning policy. When the application explicitly pins

threads sequentially, each thread has faster access to its nearest neighbors. Foreshadowing effects presented in later chapters, this benchmark experiences more memory system contention when threads are placed near their neighbors as in the thread-pinning policy. Under the memory affinity policy, the OS schedules threads to random processors. Uncontended communication takes longer because neighboring threads no longer close in the interconnection network. However, the additional queuing delay in the memory system drops because memory accesses no longer contend with one another. This contention arises in the original placement because threads frequently communicate with neighbor threads.

Most of ART's L2 cache misses are local when threads are pinned, but all of the remote misses fall on one processor. ART's average remote miss penalty to the hot-spotted node dominates the execution time. While migrating an individual thread reduces the overall memory locality, the contention to the hot-spotted node still dominates the overall cache miss penalty. Contention for the high spots is high enough that the small hot spot on node 0 becomes negligible.

More aggressive multiprocessor scheduling policies are more appropriate in a multi-user context. This section only focuses on overheads in a single-user environment. Gang scheduling incurs overheads because the OS takes longer to make global scheduling decisions and frequently schedules and de-schedules threads. The OS seeks to maximize the system throughput, not necessarily the throughput of a single application at the expense of all others. Therefore, OS developers are willing to trade some per-thread performance to ensure that all threads use the hardware to maximize system throughput.

4.3 Scheduling Overheads in a Multi-user Environment

Possible interactions of multiple processes and threads vary depending on the number of users, the memory behavior of processes that users run, and the machine size a process requests. Quantifying all the permutations of these variables is difficult.

Instead, this section presents an experiment to verify that multiprocessor scheduling policies work well in a simplified multi-user environment. The key insight is to understand how overheads measured on an unloaded multiprocessor translate to a multi-user environment where process contention is more likely.

4.3.1 Methodology

In this section's test, two parallel processes using equivalent input sets and 63 processors begin execution simultaneously and compete for system resources.

The batch-scheduled execution time is double the execution time of one process. The second process waits until the first completes before beginning execution. Batch queue scheduling is not perfect. Prior work [30] presents practical and decades-long perspective on fair and efficient batch queue policies. Fragmentation and processor starvation bottlenecks arise when many jobs of diverse sizes wait in the batch queue. Multiprocessor users should be aware of these types of bottlenecks. However, they are outside the scope of this dissertation.

For remaining multiprocessor scheduling policies, the benchmark performance should be close to double if the overheads are small. Policies that fall significantly above double fail because both users would experience a faster execution time if one user yielded the machine and waited for the other user's application to finish.

To simplify the test further, we focus only on the `SWIM` and `APSI` benchmarks, which show opposite types of locality characteristics. Almost all of the L2 cache misses access local memory in `SWIM`. This benchmark showed the largest advantage from thread pinning in the previous section. The `APSI` benchmark's L2 cache miss demonstrate almost no memory locality. This benchmark incurs almost no performance loss by leaving threads unpinned.

4.3.2 Results

Figure 4.4 illustrates the execution time of the two benchmarks across various time-sharing and space-sharing policies at 63 processors. The X-axis illustrates the three time-sharing policies, batch scheduling, gang scheduling, and run queue scheduling. Each bar represents a space-sharing policy. All execution times are normalized to the batch-scheduled thread-pinned execution time for each benchmark.

Memory-affinity and the thread-pinned benchmarks are considered to be equivalent in the unloaded tests. In the `SWIM` benchmark, the two techniques diverge when using gang or run-queue scheduling. When executing a 63-processor application, 1 processor remains idle. The scheduler on the idle processor searches other queues to find another thread to

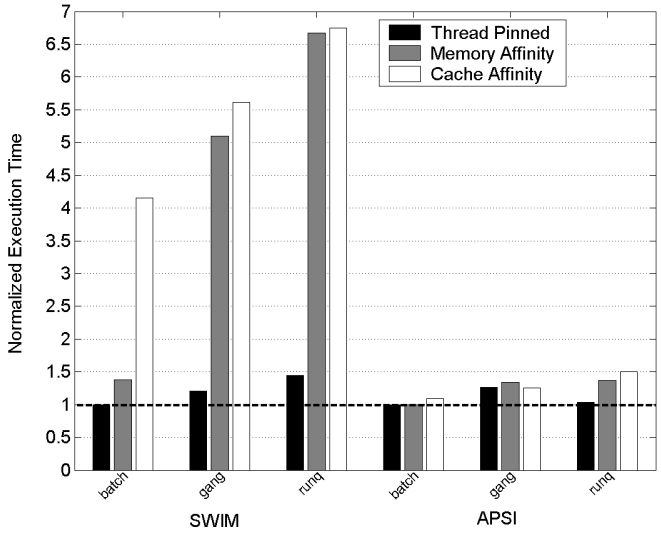


Figure 4.4: OS multiprocessor scheduling policies at 63 processors

execute and finds threads for the other process. Since the current processor is occupied executing the first process, the idle processor migrates the thread from the original node. This introduces negative thread migration and decreases performance by 5 to 7 times.

The `APSI` benchmark proves less sensitive to job scheduling because most of the cache misses are remote anyway. Beyond transferring cached data, thread migration is inexpensive. Later chapters explore software optimizations that improve the locality of this benchmark considerably. As software optimizations that improve locality are applied, this benchmark will experience a higher penalty from multiprocessor scheduling policies.

This test exposes an inherent gang-scheduling pitfall. When the first process begins execution, it spawns 63 threads and leaves one processor idle. The pitfall arises when the scheduler places the descheduled process on the idle node. Initially, the second process begins execution on the idle processor and spawns 62 additional threads on the global run queue. Surprisingly, Figure 4.4's data indicates that the scheduler's best decision is to ignore the idle node.

Under the gang-scheduling policy, the scheduler must deschedule all of first process's threads before scheduling the second process's threads on the machine. The second process

begins running on the previously idle processor until it creates parallel threads and groups all the threads into a gang. Once the second process creates the gang, all threads wait until the first process is interrupted. The parent thread reruns on the initially idle node and the scheduler places the threads on the global run queue onto individual processors—leaving one node idle. The first process tries to place all of its threads. The parent thread succeeds and migrates the idle processor. The first child fails because the remaining processors are busy.

Nevertheless, gang scheduling does improve performance over a run-queue type of system in `SWIM`. However, the run-queue scheduling overheads are unacceptably high. Gang scheduling simply makes worst-case delays slightly better.

The best solution is to precisely manage resources by giving parallel processes dedicated processors through partitioning or tiling. Properly pinning threads to specific processors avoids negative thread migration completely where possible. If batch queuing is unavailable, other time-sharing policies perform best when they are applied on a per-thread basis. Globally managing all of a parallel's process's threads does not scale.

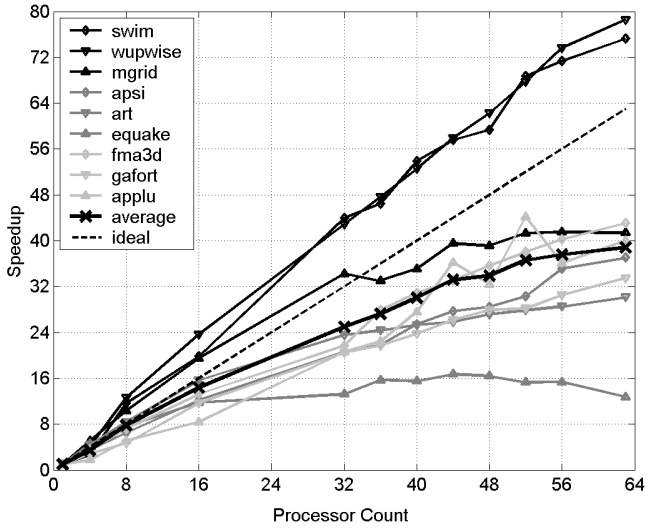


Figure 4.5: SpecOMP2001 speedup with pinning enabled and gang scheduling disabled

Figure 4.5 illustrates the SpecOMP2001 benchmark speedup curves with pinning enabled and gang scheduling disabled from 1 to 63 processors. SWIM and WUPWISE achieve near linear speedup from 16 to 63 processors. The remainder of the benchmarks with the exception of EQUAKE achieves parallel efficiencies in the range of 40%–65%. The average parallel efficiency is 85% at 16 processors, but drops to 63% at 63 processors.

4.4 Summary

Complex time-sharing techniques do not scale well to larger systems. Space-sharing techniques destroy natural locality—especially in a multi-user environment. Tight coupling between thread scheduling and data placement is required for high performance. Operating system decisions have an unusually high impact on performance at larger processor counts. Users of large-scale multiprocessors are better served by using a batch-queue scheduling policy coupled with a tiling or physical partitioning scheme.

Chapter 5

Performance Trade-offs in Memory System Design

Hardware architects focus on solving performance bottlenecks by increasing the sophistication of the memory system to reduce communication costs. These techniques include traditional methods such as caching and prefetching. In addition, architects focus on improving the cache coherency protocol to reduce the overhead of the cache coherence model. However, sophistication is not free. This chapter assesses the trade-offs between hardware techniques designed to solve performance bottlenecks and the complexity required to implement these techniques. Even ignoring the hardware costs, greater hardware complexity fails to simplify the programming model of cc-NUMA multiprocessors.

We begin by discussing the key design characteristics of coherence protocols in Section 5.1 to provide some performance metrics. To help understand how cache coherence protocols can be varied, Section 5.2 breaks protocols in to architecture, organization, and implementation layers. Section 5.3 presents the performance results of running coherence protocols implemented on FLASH with the SpecOMP2001 benchmarks and Section 5.4 looks at the performance of using a remote memory cache to extend these protocols. Because it is not practical to explore all current (or future) coherence protocols, Section 5.5 presents a limit study to show the best-case benefit an aggressive coherence protocols might provide. We find that the performance impact of the coherence protocol does not vary as widely.

5.1 Latency and Occupancy

What is the best method for analyzing memory system performance? Consider two simple properties of the memory system that the coherence protocol influences: *latency* and *occupancy*. Latency is straightforward: it is the time to send and receive data absent contention. Certain special case memory requests (e.g. multi-hop cache misses) have longer delay times than a simple request and response from the home.

Related to latency is occupancy, which is present in every system, not exclusively cache coherent multiprocessors. A general definition of occupancy is the time consumed at single-threaded control points needed to handle memory requests. When multiple requests arrive at the same time at a single-threaded control point, they contend for the same resource causing queuing delay. Examples of single-threaded control points include (but are not limited to) memory banks, the memory controller, the network, and the processor to memory controller bus. Time spent servicing requests in the memory controller and memory unit typically dominates occupancy—network occupancy is insignificant on modern shared-memory multiprocessors.

Most memory designs focus on reducing latency in the memory system. This is done by create more complex protocols, which have addition mechanisms for caching or optimizations to reduce the number of operations for specific coherency operations. More complex protocols often lead to longer occupancies, which create the potential for more contention and longer queuing delays.

Designers often explore the balance between how much latency is saved and what the memory overhead would be, but many ignore the occupancy costs because modeling contention accurately is difficult.

Simple-COMA [52] is an example of a technique that reduces remote communication by using data migration to move data closer to the CPU that is using it. Token-Based Cache Coherence [42], an example of a protocol that minimizes the overhead of cache-to-cache transfer misses. Proposals for these protocols have generally not examined the memory controller occupancy costs and focus instead on network bandwidth overheads. The flexibility in the FLASH machine allows one to implement multiple protocols and provides a valuable testing environment which includes latency and occupancy effects.

5.2 Coherence Protocols Design

The coherence protocol defines the handlers, cache line state, and messages required to provide coherent memory to the processors. There are three levels of design decisions that lead to the final coherence protocol. The *Architectural Layer* defines the coherence model, directory states and state transitions. The *Organizational Layer* defines the sharing set representation, time precision, and degree of remote caching. The final *Implementation Layer* fixes machine-specific details such as total cache and memory size, cache-line size, memory to processor transactions and I/O operations.

The FLASH multiprocessor provides a mechanism for varying the organizational and architectural layers while keeping implementation details constant. Some studies like [2, 15] compare generations of similiar architectures and organizations and evalute the impact of implementation details. This analysis is outside of the scope of this dissertation.

5.2.1 The Architectural Layer: A High Level Coherence Protocol Taxonomy

Figure 5.1 illustrates a coherence protocol taxonomy. This dissertation uses FLASH to analyze classes of coherence protocols marked in gray. Each layer in the taxonomy represents a key design decision. Higher layers have a larger impact on the final coherence protocol's latency and occupancy characteristics.

An architect's most important decision is the broad memory system architecture and programming interface provided to the user. The message passing memory system has lower occupancy compared to cc-NUMA or SMPs because there are no coherence operations to handle. Message passing machines focus solely on optimizing sends and receive and leave the programmer to explicitly program messages and handle race conditions. The cc-NUMA architecture has longer occupancies to provide coherence memory, but provides a richer programming model to the user. A symmetric memory system has the longest occupancy requirements of all because all messages are broadcast across the shared bus.

The consistency model provided to the user is a sub-layer of the memory system layer. Choosing a consistency model has a broad impact on the final coherence protocol and determines how rigidly multiple writes must be handled. For instance, in sequential consistency all writes must be removed from all other processors' caches before the requester can use

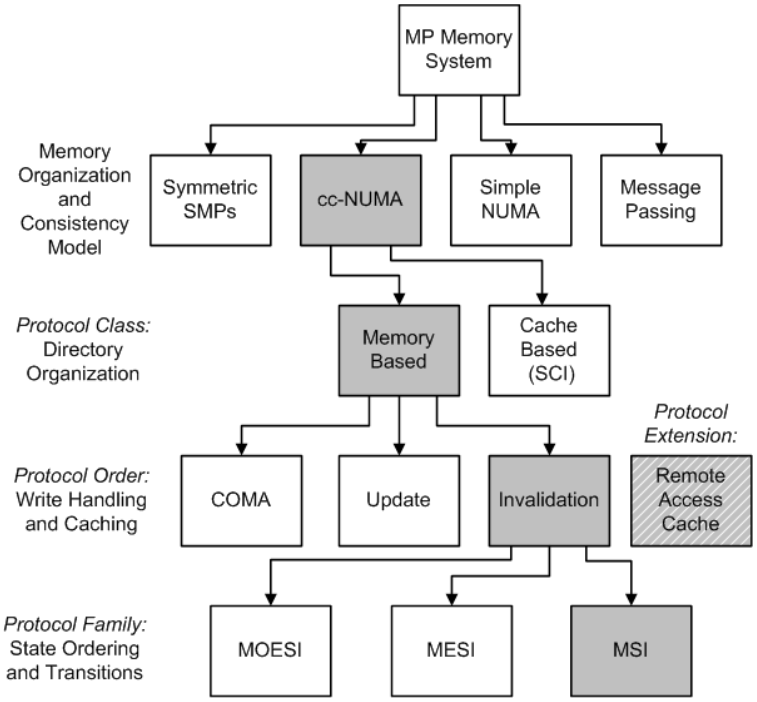


Figure 5.1: Architectural Layer taxonomy

the data exclusively. In a more relaxed consistency model, the requester could start using the data immediately once all of the invalidations have left the home. Software and hardware are often designed simultaneously. As a consequence, this decision is made early in the design process. For instance, OS developers would not look kindly on a memory system architect who relaxed the consistency model once the operating system software was complete.

The *Protocol Class* layer in the taxonomy represents the decision about where in the memory system to place cache-line state. Most coherence protocols place cache-line state in a *directory entry* in a reserved portion of main memory or a separate protocol memory closer to the memory controller. Protocols like SCI [29] store directory state in the cache with the cache tags.

In the *Protocol Order* layer, architects decide how to handle multiple simultaneous write requests. Most systems implement an invalidation-based protocol that invalidates

read copies from a processor's cache. Multiple writers must send interventions to the processor to remove dirty copies before continuing. An alternative is an update-based protocol that sends updated copies between multiple writers.

The degree of write caching is a sub-layer of the Protocol Order. The architect decides how much of memory to devote a portion of main memory to cache evicted remote data locally. This remote access cache (RAC) effectively provides a tertiary cache. Because a RAC only increases the total available cache on the system, it is considered a *protocol extension*—an orthogonal change to the layers of taxonomy. Adding a RAC does not modify the messages that are exchanged during protocol-state transitions.

Cache-only memory access, or COMA, appears to be a special case of a RAC because it treats all of main memory as cache. However, COMA changes the protocol handlers since there is no default node to store protocol state. Therefore, COMA is not a protocol extension because the messages do change during protocol-state transitions.

After deciding the Protocol Order, the read and write handling policies are well defined. At the *Protocol Family* layer, the architect chooses the precise transitions between read-only and write-allowed cache lines. FLASH uses modified (dirty), shared, and invalid (MSI) states for expressing transitions between read-only and exclusive, write-allowed data. Alternatives include modified, shared, exclusive, invalid (MESI) and modified, owned, exclusive, shared, invalid (MOESI).

From the standpoint of latency and occupancy, MSI has the highest latency-to-occupancy ratio. MSI variants like MESI and MOESI tinker with the basic state machine to optimize point-to-point latencies in cases where additional coherence operations are avoidable. For example, MESI separates the shared state into exclusive and shared states so that processors that have the only clean copy can quickly upgrade to the modified state.

5.2.2 The Organizational Layer: Lower Level Coherence Protocol Taxonomy

The protocols implemented on the FLASH machine to date are all in the MSI coherence protocol family. The lower layers of the taxonomy describe how the organizational layer decisions define the sharing list. Any of these protocols could be extended to include a RAC.

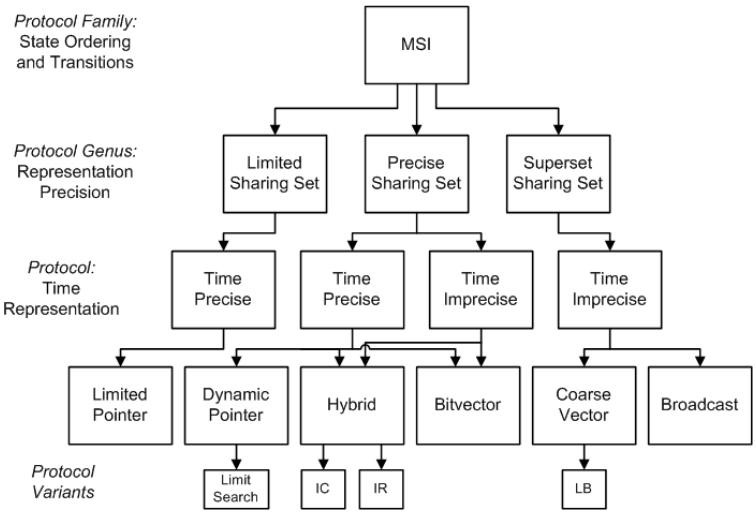


Figure 5.2: The MSI coherence protocol family

The next layer in the taxonomy, *Protocol Genus*, decides how precisely the home maintains outstanding read copy information, or the *sharing set*. The sharing set represents the list of sharers of a cache-line that must be sent an invalidation message before the home can grant exclusive rights to a requester. The sharing representation may be imprecise in two different ways. First, it may specify a superset of the actual sharers to save overhead. This may cause extra invalidations but can never cause erroneous behavior. Second, the sharing representation maybe imprecise in time, that is, it may specify sharers that no longer belong to the sharing set. Again, correctness requires that the imprecision result in too many sharers and never too few. We call the first form of precision *representation precision* and the second form *time precision*.

The key trade-offs in this layer are between memory overhead, measured in extra directory entry bits required to represent the sharing set, the memory latency of write operations, additional network bandwidth to communicate sharing information, and the occupancy overhead of using and maintaining the sharing sets.

The representation precision describes how accurately the directory entry's sharing set reflects the actual set of read-only copies held in remote caches. In a *superset representation*, a protocol may hold a superset of the sharers to reduce the bits in the directory required

to hold the sharing set. When a requester needs exclusive rights, the home sends unnecessary invalidations to processors that have not requested a copy of the cache-line. A *precise representation* means that all sharers in the sharing set reflect requesters that have had the cache-line since the last invalidation operation. In the third case, *limited representation*, the directory entry can only hold a fixed number of the current sharers of a cache-line. If a requester arrives when the directory entry is full, the home invalidates one of the current sharers to service the new requester.

The second level, called time precision, describes how accurately the sharing set represents sharers in time. For example, this type of protocol precision specifies what the protocol must do if the cache evicts a read-only copy of a cache-line. In a *time imprecise* protocol, a remote node need not inform the home that it has evicted the read-only copy. During a write operation, the home sends unnecessary invalidations to any node that has already evicted their read-only copy. In a *time precise* protocol, a remote node always sends a *replacement hints* to the home when the L2 cache evicts a cache-line. These messages, sometimes called *eviction notices*, decrease the latency of write operations because unnecessary invalidations are seldom sent. However, all read-only cache evictions cause replacement hints.

Even protocols that use replacement hints cannot be strictly time precise. Occasionally, the home sends an invalidation message to a remote node for a cache line that has already been evicted from the remote node's cache. The time imprecision occurs if the replacement hint has not arrived at the home before the initial invalidation leaves the home.

After making decisions on sharing set precision, architects decide how to organize the sharing set in the directory entry to determine the final protocol. A limited representation requires precise timing information to keep the sharing set as small as possible. A precise representation of the sharing set may require replacement hints, if the memory overhead depends on the number of sharers in the sharing set. A protocol that uses a superset representation cannot effectively use timing information to remove sharers because the sharing representation represents multiple nodes at once.

For example, the simplest representation of the sharing set is to use a *broadcast* protocol that invalidates a cache-line in every cache, regardless of whether it is a sharer or not. In a FLASH-like architecture, this solution would consume enough network bandwidth to significantly affect scaling beyond even a few processors. Once any processor shares the

cache-line, the protocol assumes all processors share the cache-line and the home must send an invalidation message to all caches during a write operation. If a node sends a replacement hint to the home, the sharing set does not maintain enough fidelity to reduce invalidations because the directory entry must still assume the cache-line is shared globally.

Bit-vector Protocol

The simplest practical representation for the sharing set is to use a bit-vector where every bit in the vector corresponds to a node in the system [13]. Handlers have short occupancies since the control is straightforward requiring simple control logic. This protocol has a precise representation, but introduces costly memory overhead, which scales quadratically with the processor count. Bitvector could be either time precise or imprecise because it is uses a precise representation.

One common technique to lower the memory overhead is to relax the representation precision. A *coarse-vector* protocol overloads one bit in the vector to represent multiple nodes topologically close to one another [25]. While reducing the overhead, this protocol introduces longer write latencies to service extra invalidations. The number of processors that one bit represents the degree of coarseness. We name the coarse vector protocol using the convention $CV = n$ where n represents the degree of coarseness. Therefore, bitvector is a $CV = 1$ protocol and broadcast is a $CV = P$ protocol. Because coarse vector uses a superset representation, this protocol does not maintain timing precision.

Dynamic Pointer Allocation Protocol

An alternative approach that keeps a precise representation is to use a linked-list to store the sharing sets. The link pointers are dynamically allocated from a fixed-size link-store, therefore this protocol is known as dynamic pointer allocation [55]. Each sharer in the sharing set is a *link element* in the linked-list. The advantage of dynamic pointer over bit-vector is that the average number of sharers times the number of shared cache lines determines the memory overhead—not the rare worst case where all cache lines are shared everywhere. If a new sharer needs to be added when the *common link-store* is full, the handler invalidates a random list to free up link elements. This process is called *pointer reclamation*.

Dynamic pointer allocation protocol implementations only work well if they are time precise. Replacement hints keep the sharing list size small and reduce the need for pointer reclamation, which is expensive. When adding a sharer, there is no check to see if a sharer is already on the list to keep GET operations low-occupancy. The replacement hint handler spends the occupancy cost of traversing the linked-list to remove the evicted sharer, which execute after cache misses are satisfied. The dynamic pointer allocation protocol sends fewer invalidation messages compared to coarse-vector, because the sharing list is precise in representation and time.

However, replacement hints cost more occupancy as the potential size of the linked-list grows with processor counts. To solve this problem, the replacement hint only traverse the first few elements of the list (our implementation uses 16) to see if the element is on the list. This technique is called *limit search*. Point-to-point latency is not affected by replacement hints since they occur after the cache miss is satisfied. However, they do occupy the memory controller and can potentially contribute to contention.

A simpler form of the dynamic pointer protocol keeps only a limited number of sharers for one cache-line at a time. This technique is called *limited pointer*. The key disadvantage is that this protocol uses limited representation. If a cache line is frequently shared by more nodes than available in the limited pointer, extra misses occur due to the frequent unnecessary invalidations as sharers are evicted from the limited pointer. Frequently accessed globally shared data, as an example, would frequently incur unnecessary cache misses. To reduce these extra cache misses, limit pointer must be kept time precise.

Hybrid Protocol

We implemented the hybrid protocol after realizing that the occupancy cost of replacement hints limits large-scale multiprocessor performance (presented in simulation [26]). Our goal was to create a protocol that precisely represents the sharing set, like bit-vector, with the advantage of bounded memory overhead, like dynamic pointer, without requiring time precision for performance. We call this protocol a *hybrid* because it uses mechanisms from both. The hybrid protocol is a two-level protocol, which uses multiple definitions of the sharing list depending on the number and position of remote sharers. This protocol could maintain time precision if necessary because the sharing set maintains a precise representation.

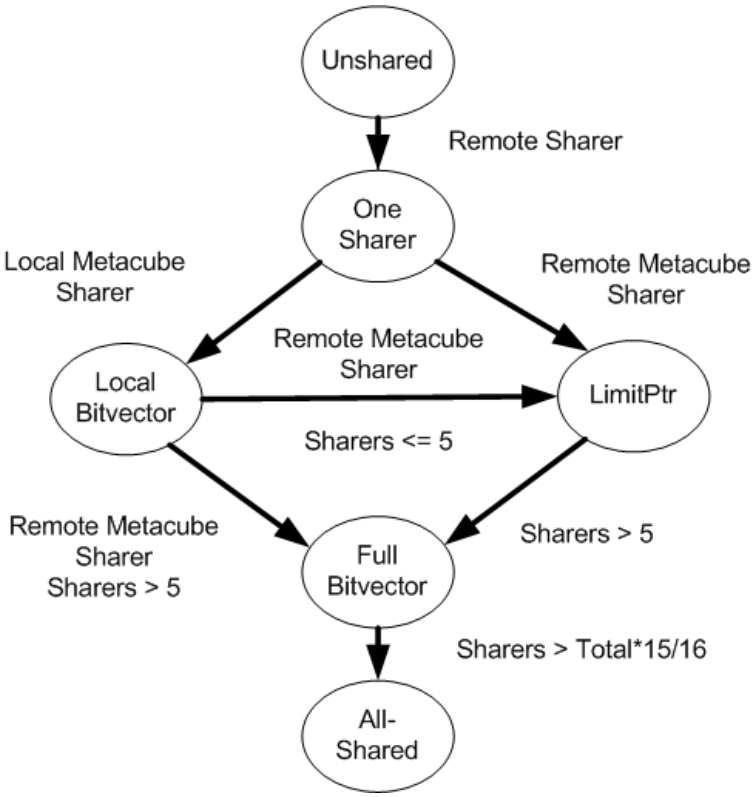


Figure 5.3: Sharing list state machine of sharing state list

Figure 5.3 presents the state machine for the sharing list. The initial state for all cache-lines is UNSHARED. When any cache-line is written, any current sharer is invalidated. Then, the directory entry is set back to this default state. The bit formats for each additional directory-header state are illustrated in Figure 5.4.

The first sharer moves the state machine into the ONE_SHARER state. There are many degenerate coherence cases in which no invalidations are required and writes can be immediately satisfied. Every protocol checks for these degenerate cases by testing if the sharing set is empty (no sharers) or only shared by the requester (one sharer). Implementing a separate state for one sharer simplifies these checks. Once a second remote sharer is added, the protocol then decides on a new representation for the sharing set.

If the sharer is in the neighborhood local to the home, the sharing set transitions to

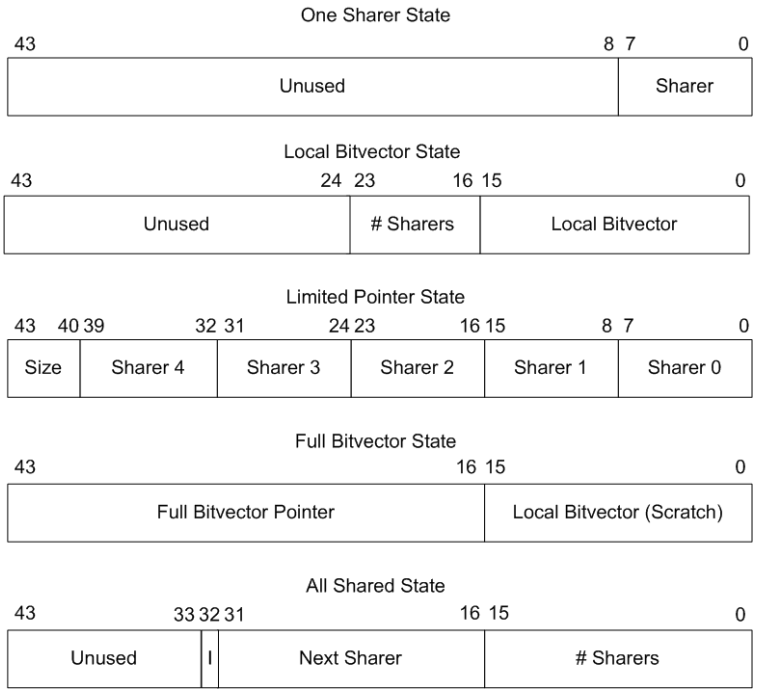


Figure 5.4: Bit definitions of sharing state list

the LOCAL_BITVECTOR state. The neighborhood is a subjective term. FLASH is partitioned into metacubes of 16 nodes. Latencies to remote metacubes are almost double latencies to the local metacube. So neighborhood is defined as metacube, and nodes on the same metacube are represented by this local bit-vector. Bits are also reserved to represent the total number of sharers. As long as any new sharer added in this state is on the local metacube, the directory header stays in the local bit-vector state. If the new sharer is on a remote metacube, the directory header needs to switch states since it no longer has enough bits to represent the sharing set. If there are fewer than 6 sharers, the directory moves to the LIMIT_PTR state. If there are more than 5 sharers, the state changes to FULL_BITVECTOR.

If the new sharer is on a remote metacube, the sharing set transitions to the LIMIT_PTR state, which keeps up to 5 sharers. The hybrid protocol does not suffer from the traditional problems present in protocols with limited sharing lists because the protocol transitions into the FULL_BITVECTOR state if the sharers exceed directory header capacity.

The FULL_BITVECTOR state retains some key features of dynamic pointer. Like the pointer-link store, this protocol has a bit-vector store. Whenever a directory header moves into the full bit-vector state, the handler allocates a long bit-vector with enough bits to enumerate all of the processors in the system. If the common bit-vector store is full, the handler picks one long bit-vector at random and reclaims it by sending invalidations to free an entry.

Once the total sharers count crosses a threshold, the directory header state changes to ALL_SHARED. This transition occurs at 60 processors or 15/16th of the entire machine size, but can easily be changed by raising or lowering the transition threshold. Reclamation happens less frequently because more long bit-vectors are available. The costs of this state are at most four extra invalidations if the cache-line is not globally shared. The total sharers are stored in the original directory header along with some book keeping variables, (e.g. "Next Sharer") used when all-shared lines are invalidated.

Occasionally, a sharer requests exclusive access to globally shared data. If globally shared data is written frequently, only a few sharers will likely get a read-copy at a time. Even so, frequently upgrading global-shared data is a pathological case that would require an application rewrite to remove this case. To handle this correctly, however, the directory header transfers back to the FULL_BITVECTOR state and does not send an invalidation message to the requester.

The hybrid protocol uses a `Sharing Set State` (bits 44 through 47) to determine the sharing list format. Therefore, any time the coherence protocol needs to do some book-keeping on the sharing set, it looks into a state table to do a special operation depending on the sharing list state.

5.2.3 Protocol Variants

Some changes to coherence protocols take the base state-machine and modify it in some minor way without changing the protocol's precision. These protocol variants seek to reduce the overhead of remote communication latency in general or the latency or occupancy observed in special cases unique to the cache-coherence model. A protocol variant retains the higher-level architecture and organizational characteristics of a protocol but changes how protocol transitions are handled.

Standard Variant

First, we need to describe how MSI protocols behave when transitioning to the modified state from the shared state under sequential consistency. Normally, a requester sends an UPGRADE or GETX request to the home. The home's protocol processor then iterates through all of the sharers in the sharing set, sending an invalidation message to each. When remote sharers receive invalidation messages, they immediately send a reply back to the home before sending an invalidation to the processor. The home collects all of the invalidation acknowledgments before sending a reply back to the requester. Once the original requester receives the reply from the home, it can begin using the data exclusively.

Write latencies under sequential consistency are longer than more relaxed models. For example, the home could send a reply back to the requester before invalidating the sharing set. This technique violates sequential consistency because read-only copies of a cache-line coexist in time with an exclusive copy. Each memory controller does the same amount of work, only in a different order for more relaxed models. Therefore, the occupancy of the invalidation operation remains the same independent of the consistency model.

Coarse-Vector with Local Bit

This technique seeks to improve the representation precision, as in hybrid, while maintaining the low-occupancy handlers present in the coarse-vector protocol. In $CV = 2/LB$, the coarse-vector protocol holds a special local bit to indicate that only the local processor holds a read copy of the cache-line. This optimization introduces slightly longer occupancies in handlers that check if a cache-line is unshared because both the local bit and the coarse vector must be checked.

Clustered Invalidation Protocol Variant

One question is whether spreading occupancy around the system would help balance occupancy and reduce the overall cost of queuing delay. In clustered invalidations, labeled $Hybrid/IC$, handler occupancies are shifted from the home to fairly balance occupancy costs among remote sharers.

One cause of communication hot spots is bursts of invalidations sent and collected by the home when a cache-line is shared among many nodes. The home sends all of

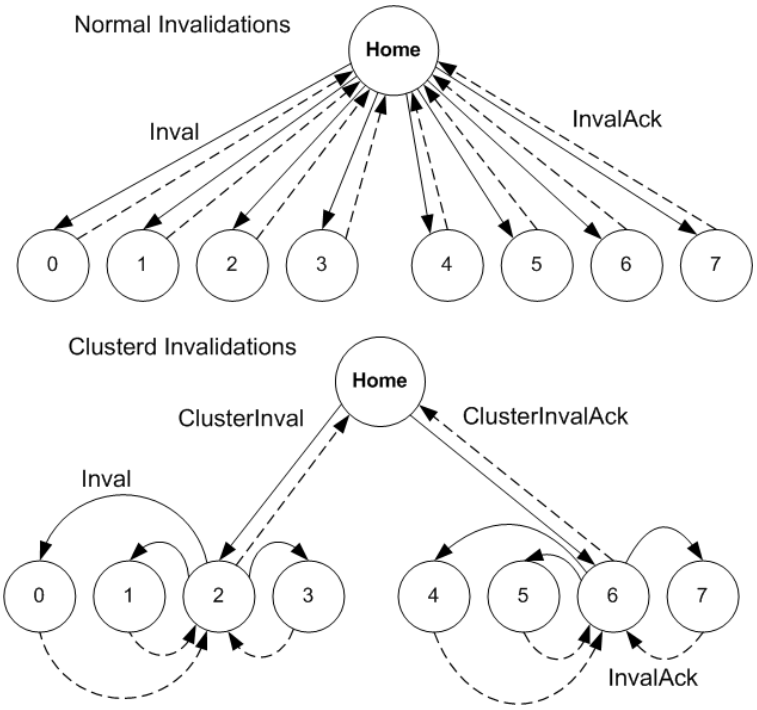


Figure 5.5: Normal versus clustered invalidations

the invalidations and collects all of the invalidation acknowledges. Therefore, the home experiences a larger occupancy cost than the requester or the individual sharers.

The *clustered invalidations* variation, illustrated in Figure 5.5, shifts occupancy away from the home. When many invalidations are required, the home sends invalidation cluster messages, `MSG_CLUSTER_INVAL`, to nodes topologically close to one another. Nodes that receive a `MSG_CLUSTER_INVAL` send invalidations to nearest neighbor nodes and then send a clustered invalidation acknowledge, `MSG_CLUSTER_INVAL_ACK`, back to the home. The extension sets special bits in the message header to filter out nodes in the cluster that do not require invalidations.

These cluster messages reduce the number of messages that the home has to send and receive and balances the occupancy among all sharing nodes. The latency of large invalidations absent contention is slightly longer because there is now a tree structure to invalidation messages. In theory, this extension reduces hot spots, thus lowering the overall queuing delay.

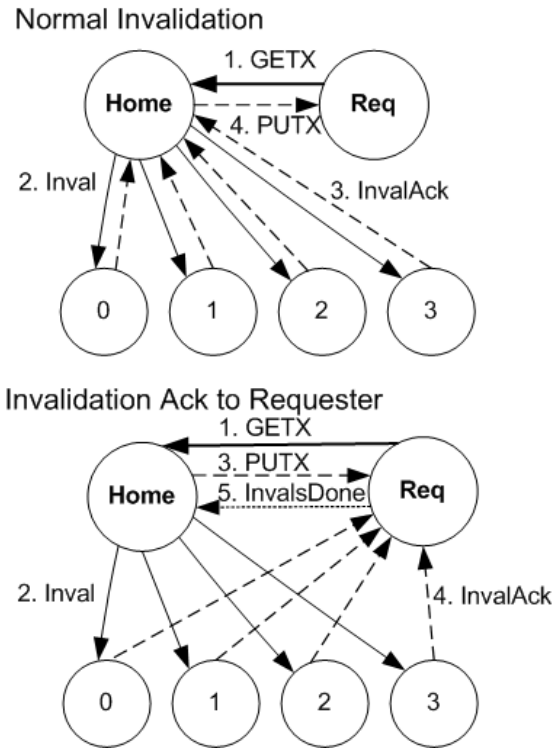


Figure 5.6: Base versus requester invalidations acknowledgments

InvalAck-to-Requester Variant

The second technique, called *InvalAck-to-Requester*, moves occupancy away from the home by collecting invalidation acknowledgments at the requester. This technique, labeled $Hybrid/IR$ lowers the occupancy cost and the queuing delay at the home during invalidations. There is also a latency advantage to collecting invalidation acknowledgments at the requester because the home immediately sends the data back after sending invalidations. Once the data arrives from the home, the requester immediately uses the data exclusively without waiting for all of the invalidation acknowledgments to arrive. Obviously, this technique is meaningless if the home and the requester nodes are identical.

A typical `GETX` transaction is illustrated in Figure 5.6 for both cases. Normally, the requester sends a `GETX` request to the home. The home then sends invalidations out to all of the sharers. Once all of the invalidations acknowledgments return, the home sends the

data back to the requester using a PUTX message. If the cache-line is globally shared, the invalidation time can be considerable. However, all of the invalidations could be serviced in parallel if the home broadcasts out all of the invalidations at the same time. In practice, this is difficult to do since outgoing messages contend for access to the network.

Collecting invalidations at the requester allows the home to send the data immediately after sending out all of the invalidation messages. Identical to the base case, the home receives a GETX request from the requester and sends the invalidations. However, the home sends a PUTX once all of the invalidations are sent. In addition, the home tells the requester how many invalidation acknowledgments to collect. The requesting MAGIC receives the data and sends it to the main processor core and records the cache-line in an invalidation collection table. To avoid race conditions when multiple requesters attempt to write the cache-line at the same time, the home keeps the cache-line pending. The requester sends a MSG_INVAL_ACK_COLLECT_DONE message back to the home once all of the invalidation acknowledgments arrive. The home resets the cache-line pending bit when this message arrives.

This extension requires additional memory to hold the invalidation collection table. In our experience, a table size of 16 is sufficient. If multiple requesters frequently request exclusive access for the same cache-line, the latency for each write will be longer because the cache-line will be held pending for a longer period of time.

This variant slightly relaxes the consistency model because the requester may start writing the cache line while some read copies still exist in other processors' caches. Violations of sequential consistency occur when the invalidation acknowledgments loses the race with the PUTX by enough time for the processor to resume execution and modify the data. Invalidation acknowledgments have a slight advantage because the home sends the invalidations first before sending the PUTX to the requester. In practice, the number of sharers is usually small because the home keeps a precise representation. In this common case, most of the invalidations win the race to the requester before the reply arrives from the home. This consistency model is strict enough to boot IRIX and execute our benchmarks correctly. However, if the home sends the PUTX first before sending the invalidation, IRIX fails to boot properly. The invalidations need the head start to commonly win the race to the requester.

Table 5.1: Coherence Protocol Complexity and Size

Protocol	All Handlers		Critical Handlers		I$ Statistics (63p)	
	Size (KB)	*Ratio*	*Size (KB)*	*Ratio*	*Miss Rate (%)*	*Occup (%)*
CV=2	130.17	1.00	8.68	1.00	0.0050	0.2236
CV=2/LB	130.38	1.00	8.92	1.02	0.0222	0.6241
DynPtr	140.66	1.08	19.83	2.28	0.1546	3.6126
Hybrid	149.91	1.15	18.65	2.14	0.0724	1.5676
Hybrid/IC	150.49	1.16	19.63	2.26	0.1515	4.0131
Hybrid/IR	156.18	1.20	23.18	2.67	0.1599	4.1581
Hybrid/RAC	157.58	1.21	23.13	2.66	0.6231	13.2131

5.2.4 Coherence Protocol Complexity

FLASH implements coherence protocols as firmware programs loaded into the MAGIC embedded processor during the machine boot. Extra protocol complexity translates into longer handler occupancy, measured by additional instructions in the protocol program.

The coherence protocols must be carefully optimized to avoid the FLASH-specific bottleneck created by the small size of the MAGIC's instruction cache. Only the subset of protocol handlers related to the most common memory operations influence the overall program. The sizes of the most critical handlers must remain close to 16KB to minimize the impact of protocol instruction cache misses.

Table 5.1 summarizes the program sizes of each of the coherence protocols and protocol extensions used in the remainder of the chapter. Column 2 presents the overall size of the coherence protocol program. The third column normalizes the sizes to the simplest $CV = 2$ protocol. Columns 4 and 5 present the total and normalized size of the critical set of handlers for each protocol. Columns 6 and 7 detail MAGIC's instruction cache miss rate and the percentage increase in overall occupancy. Each instruction cache miss increases latency by 87 processor cycles if the misses occur during the latency path of the handlers.

This table also presents the additional complexity required to implement the protocol variants and the RAC extension. $CV = 2/LB$ only increases the size of the protocol program by 2%. The $Hybrid/IC$ extension increases the size of critical handlers by 5.6%. The $Hybrid/IR$ extension increases the size of the critical handlers by 25%, which is considerably larger than the other extensions. This increase results from extra handlers

required to implement remote invalidation acknowledgment handlers, which are not necessary in the base hybrid protocol, and additional complexity to maintain an invalidation acknowledgment table on each requester.

While the hybrid's and dynamic pointer's critical handlers are more than twice the complexity of the coarse vector handlers, the additional complexity does not generate abnormally large instruction cache misses on MAGIC. Added instructions in individual handlers do influence the latency and occupancy characteristics of the protocol, which we consider separately in Section 5.3.

In comparison, the RAC protocol extension begins to suffer from poor instruction cache behavior. While the critical size of the handlers is consistent with the other protocols and extensions, the occupancy increases by 13% due to these extra instruction cache misses. This extra occupancy is not related to expected increases in occupancy to check the RAC tags. MAGIC does not provide statistics on which instructions in the firmware cause cache misses. As a result, we cannot extrapolate how this FLASH-specific occupancy translates into longer cache access times.

Section 5.4 explores the trade-off between latency and occupancy in the RAC in more detail. On an idle machine, the benefits of remote caching outweigh the extra latency incurred by extra instruction cache misses at least on an idle machine. So we expect that while RAC has longer occupancies relative to other protocols and protocol variants, the remote latencies when using a RAC should drop despite poorer instruction cache behavior.

5.3 Quantifying Realistic Coherence Protocol Behavior

This section presents our experiences with coherence protocols on the FLASH machine and explores the practical limitations of removing performance bottlenecks in hardware. Coherence protocols balance performance of applications against the challenges in implementing the protocols in hardware.

We would expect that performance differences between the protocols would track changes in the point-to-point latency of cache misses because each protocol handles memory requests in a similar fashion. To explore our hypothesis, this section first presents a point-to-point latency analysis of the three base protocols. Using the SpecOMP2001 benchmarks, we find that relative performance differences of the applications do *not* track the

relative differences in point-to-point latency at larger processor counts. Occupancy introduces queuing delay causing significant slow-downs in the more complex protocols.

Could a protocol effectively manage occupancy using the protocol variants that migrate occupancy away from potential hot spots? To answer this question, the section considers the performance of the protocol variants discussed in the previous section at higher processor counts. What we see is that the protocol variants in most cases only hurt performance because the complexity introduces significant contention and subsequent queuing delay that dominates the hot spot.

5.3.1 Point-to-Point Latency Analysis

A natural starting point when analyzing different coherence protocols is to measure point-to-point latency. While each of the base protocols is similar, one does observe minor latency differences. This sub-section quantifies those differences using the snbench tool [50] developed for analyzing memory latencies on SGI's Origin 2000.

We use the micro-benchmark results presented in this section to understand contention in the high-level SpecOMP2001 benchmarks. If contention is small, as it is on small-scale machines, then the benchmarks' relative performances should track measured relative differences in point-to-point latency.

Read Latencies

Local miss latencies absent contention are identical for each node in the system. The latency of remote read misses and local read misses requiring remote action depend on the physical distribution of the copies of the cache line and particular coherence protocol. We calculate best-case and worst-case read latencies for remotely placed data. The best case is remote data on the same 4-node cluster. Worst-case latencies occur when messages are routed through the upper meta-cube mesh to a node on a different meta-cube.

Table 5.2 summarizes the read latency data provided by snbench. The local read times are identical. Similarly, there is little variation in remote read times. Adding a sharer to the sharing set costs occupancy, not latency, in this test since the PUT is sent before the handler adds the sharer to the sharing set. There is considerably more variance in reading remote dirty data. The hybrid protocol is slower than coarse-vector by 18.4% in the minimum

Table 5.2: Protocol Read Latencies in Clock Cycles and Normalized to CV=2

Protocol	*Local Read*	*Ratio*	*Remote Read*	*Ratio*	*Three-Hop Read*	*Ratio*
CV=2	135	1	303-483	2.24-3.58	588-720	4.13-5.33
DynPtr	135	1	303-489	2.24-3.62	645-720	4.78-5.33
Hybrid	135	1	306-483	2.27-3.58	696-909	5.15-6.73

latency case, and 26% in the worst case. The extra delay is due to longer handlers that must determine the directory entry state at the home before forwarding the `GET` request to the remote dirty node.

If three-hop read misses are rare, the three protocols should share identical read point-to-point latencies. If three-hop read misses are frequent, in the absence of contention, we expect hybrid to perform worse relative to dynamic pointer or coarse-vector, which should be similar to each other.

Write Latencies

The `snbench` tool includes a test to measure upgrade bandwidth rates. However, this test by default does not correspond directly to upgrade latency because the R10k allows up to 4 outstanding memory transactions at a time. We reconfigure the R10k to only allow one outstanding transaction at a time. Therefore, the upgrade latency is the time required to upgrade 128B of data, the cache line size of R10k's L2 caches. The remote sharer count varies from 1 to 60 nodes. `Snbench` uses a monitoring thread to calculate upgrade bandwidth, and above 60 processors the monitoring thread frequently is scheduled on a sharing node, inflating the observed latency. The test runs `snbench` multiple times with random sharer distributions to yield an average upgrade latency.

The upgrade bandwidth test ignores two write cases: remote misses that cause remote invalidations and remote misses that cause three-hop, cache-to-cache transfer write misses. In the former case, the extra hop is much smaller than the invalidation time. The latter case is dominated by the processor intervention time—identical across each protocol.

Figure 5.7 graphs the average upgrade latency versus the remote sharer count for the three coherence protocols from 1 to 60 remote sharers. Most writes in SpecOMP2001 benchmarks invalidate 1 to 16 remote copies, therefore Figure 5.8 illustrates this range

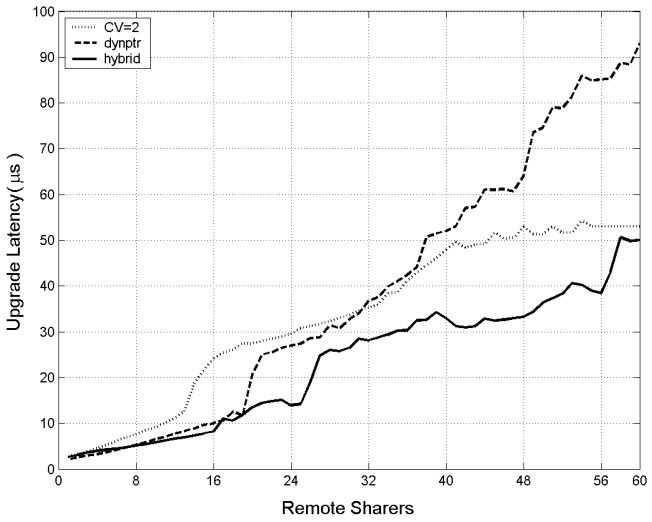


Figure 5.7: Average upgrade latency versus total sharers

in more detail. The upgrade latencies are high enough relative to the extra time to fetch data from memory for a GETX request that writes and upgrades can be considered to share identical latencies with little error.

For the dynamic pointer and hybrid protocols, the remote sharers track with the total invalidations sent during the test. In the coarse-vector protocol, the total invalidations sent vary depending on the remote sharer distribution. For example, if only odd requesters share a cache line, extra invalidations are sent to the unshared even node.

Dynamic pointer has the shortest average upgrade latency at small remote sharer counts (1 to 7). Sending invalidations takes longer than the other protocols at higher processor counts since each pointer in the linked-list must be read before invalidating the next sharer. Longer lists are also less likely to be cached by MAGIC. Above 16 processors traversing the list can take longer if there are duplicate entries due to the limit search algorithm used during replacement hint handlers. Global invalidations take almost twice as long as the coarse-vector and hybrid protocols.

The hybrid protocol has longer latencies at smaller remote sharer counts due to the 2nd-level directory state lookup required before sending invalidations. At higher remote

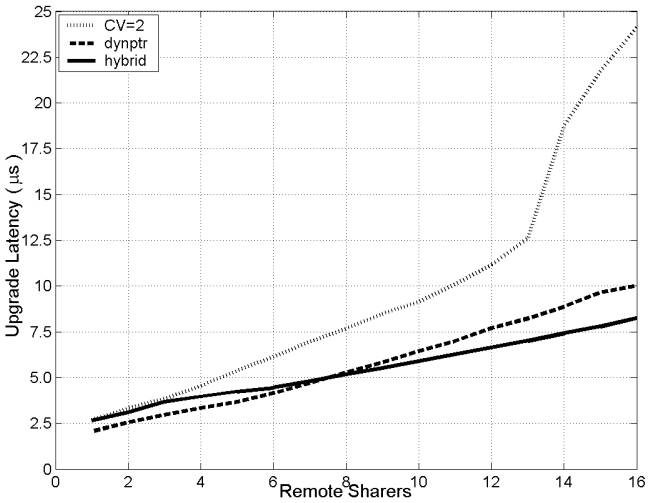


Figure 5.8: Average upgrade latency from 1 to 16 sharers

sharer counts, hybrid's simple bit-vector representation takes less time to invalidate since it does not have to traverse a linked-list as in dynamic pointer and benefits from a precise representation of the sharing set.

The micro-benchmark write latency tests indicate that there are significant differences in large-scale invalidation times. To understand relative differences in the protocols that arise because of invalidation latency differences, we augment the coherence protocol to record the distribution of invalidations per write. When a write-miss occurs, each coherence protocol records the total invalidations sent. Multiplying the total writes and upgrades by the memory latency for a given average sharers per write yields the aggregate invalidation time. Of course, using the aggregate invalidation time to explain absolute differences between protocols proves is difficult because multiple outstanding requests hide some write latencies.

The total occupancy, measured by summing the occupancy of every node, is a more appropriate metric than per-node average or maximum occupancy of one node. Occupancy penalizes performance by increasing contention around hot spots. Hot spots cause other nodes to stall, decreasing the average occupancy observed for the machine, even though

the total occupancy increases [26]. Even using maximum occupancy is misleading because the hot spots may migrate or disappear as a benchmark's phase changes. The total occupied cycles measures directly the impact of protocol occupancy on performance.

A protocol extension that successfully improves performance by providing shorter point-to-point latency at the expense of occupancy should show a rise in total occupancy cycles, but should show a drop in the total execution time. We measure total occupancy in the system by summing occupied cycles across every node in the system. Results presented later in this chapter show that the opposite trend is true: there is a strong correlation between high total occupancy cycles and lower performance.

5.3.2 SpecOMP2001 Protocol Results

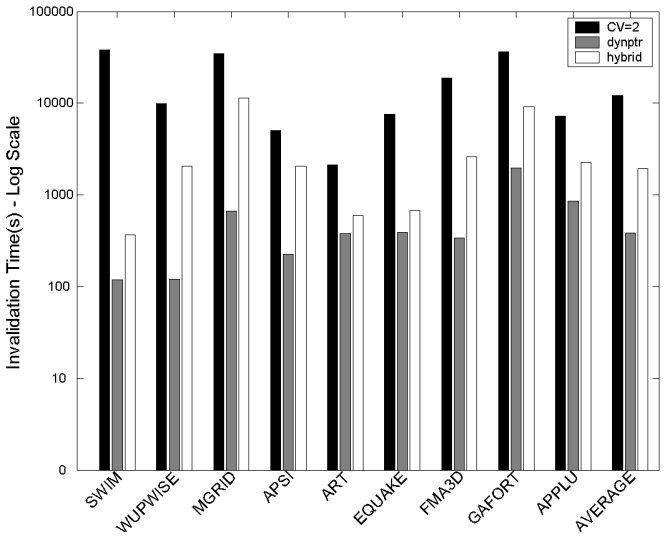


Figure 5.9: Aggregate invalidation time at 63 processors

Figure 5.9 demonstrates that there are large variations in invalidation time across the benchmarks. Coarse-vector on average spends 31 times longer than dynamic pointer sending invalidations! The problem with coarse-vector is that most local upgrade requests must

also send an invalidation to their nearest neighbor due to bit coarseness. Across all benchmarks, dynamic pointer consistently spends less time doing invalidations. This is expected: the replacement hints in dynamic pointer maintain more precise sharing information.

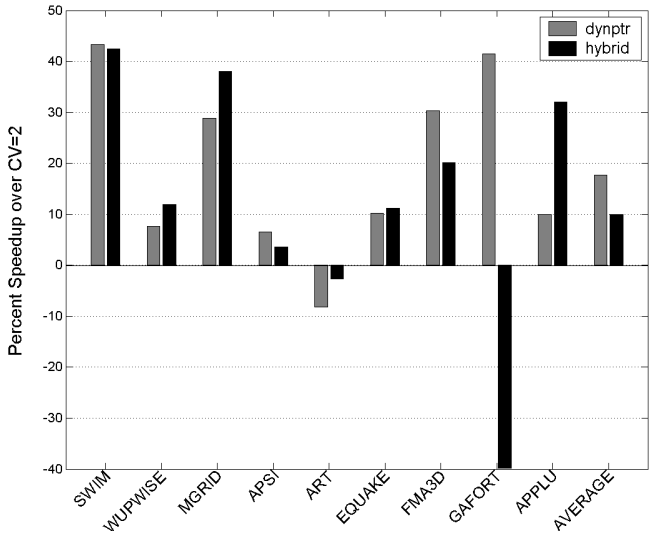


Figure 5.10: Speedup relative to the CV=2 protocol at 8 processors

Given this invalidation data and the relative similarity of the read latencies, we expect dynamic pointer to outperform hybrid, which in turn will outperform coarse-vector. At 8 processors, the predictions based on uncontended latency track overall performance. Figure 5.10 shows the percent speedup of the parallel sections of the SpecOMP2001 benchmarks over the coarse-vector protocol. Dynamic pointer outperforms coarse-vector by 17% and hybrid by 6%.

In `WUPWISE`, `MGRID`, `ART`, `EQUAKE`, and `APPLU`, the dynamic pointer runs complete faster than the hybrid pointer runs at 8 processors because the dynamic pointer protocol maintains a time-precise sharing set. The coarse-vector protocol is 66.7% faster than the hybrid protocol in the `GAFORT` benchmark where there are fewer invalidation messages per write. In `ART`, coarse-vector is faster than hybrid and dynamic pointer. In this benchmark, the performance bottleneck is unrelated to the precision of the sharing set. A hot spot exists at the master node causing occupancy to impact performance by creating contention.

The remaining applications behave as expected given the point-to-point latency differences measured with the micro-benchmark tests.

However, the protocols perform quite differently at 63 processors. While dynamic pointer consistently sends fewer invalidations during write operations, the queuing delay caused by long occupancies in replacement hint handlers removes any performance advantage (an effect not seen at 8 processors). This result demonstrates that at larger machine sizes, minimizing occupancy costs may prove more important than maintaining time-precise sharing sets and shows the limitation of using small machine sizes to validate coherence protocol proposals.

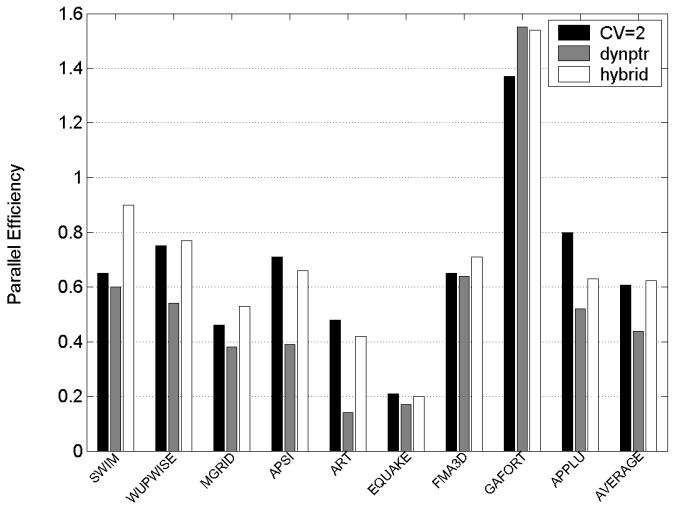


Figure 5.11: Parallel efficiency at 63 processors

Figure 5.11 presents the SpecOMP2001 benchmark data for parallel efficiency. There are significant differences among benchmarks between coarse-vector, dynamic pointer, and hybrid protocols. Figure 5.12 illustrates the relative percent change in performance over coarse-vector. Dynamic pointer, which has the best invalidation performance, has the worst overall performance. Hybrid and coarse-vector protocols have more comparable performance—on average only differing by about 2%. The two protocols behaviors, however, benefit from different effects. In coarse-vector, the lower occupancy demands of the

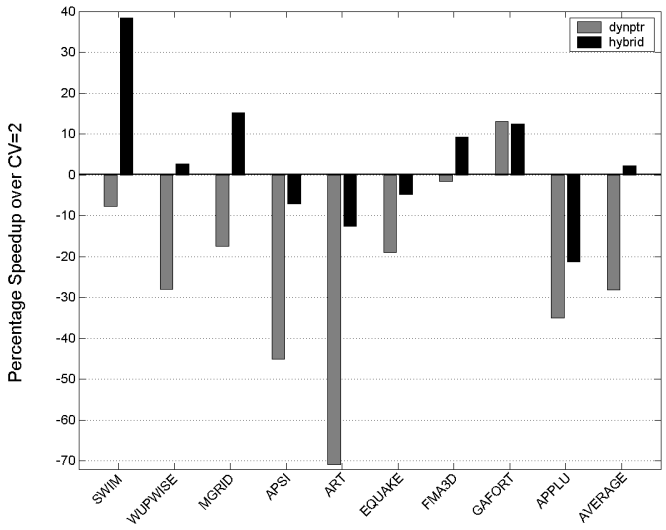


Figure 5.12: Speedup relative to the CV=2 protocol at 63 processors

handlers reduce contention, but the extra invalidations increase the latency and occupancy of write operations because the sharing set is not precise in representation. Hybrid sends fewer invalidations by keeping a precise representation. The precision keeps latency and occupancy of write operations smaller, but longer handler occupancies are needed to keep the two-level directory state and manage transitions between sharing set states, which offset some of the precision advantages. Overall, the design trade-offs made in the coarse-vector and hybrid protocols capture similar performance improvements over dynamic pointer.

Individual benchmarks have wider variations. For instance, in `SWIM` the hybrid protocol speeds up by 40% over coarse-vector. For `APSI`, `APPLU`, and `FMA3D`, hybrid is slower than coarse-vector. `GAFORT` is one case where dynamic pointer's parallel efficiency is on par with hybrid and outperforms coarse-vector by 13%. Remember, at 8 processors, dynamic pointer is 2.22 times faster than hybrid. The shorter time for sending invalidations in dynamic pointer combined with precise sharing information is a significant advantage at 8 processors. However, at 63 processors, the dramatic performance benefit of dynamic pointer at 8p is lost due to queuing delay. As machine size continue to scale beyond 64 processors, hybrid will likely outperform dynamic pointer for this application.

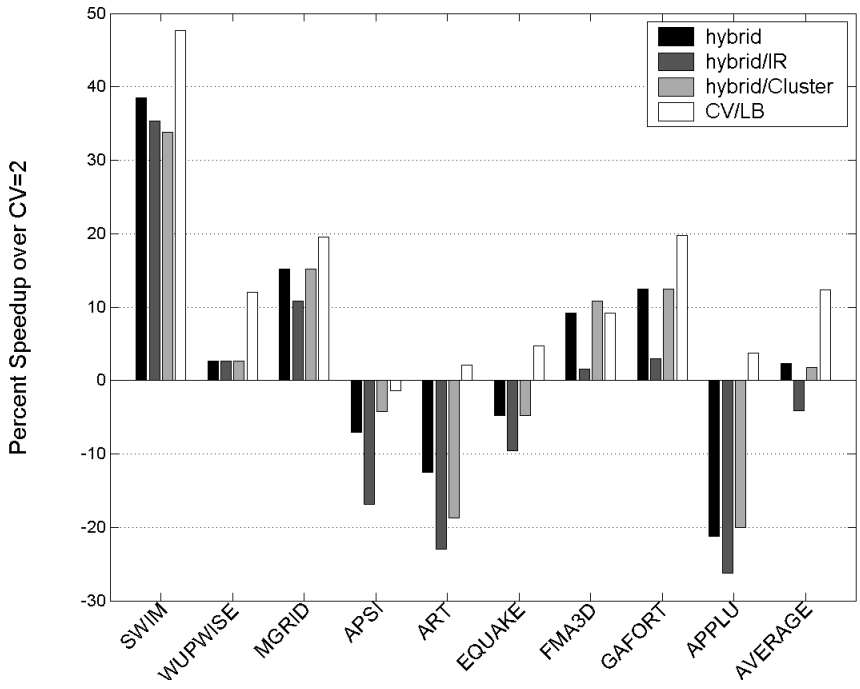


Figure 5.13: Speedup relative to CV=2 at 63 processors

5.3.3 Protocol Extensions

This section considers three specific protocol extensions and their impact on parallel efficiency. Figure 5.13 presents the relative effect of these protocol enhancements over the base coarse-vector protocol. The hybrid bar represents the original hybrid protocol.

For most benchmarks, the $CV = 2/LB$ outperforms the base $CV = 2$ by 10%, because most of unnecessary invalidations due to representation imprecision are removed. Local upgrades do not need to send an invalidation message to the nearest neighbor unless it is a real sharer. This optimization also reduces the queuing delay by lowering the arrival rate of messages. `APSI` does show a small performance loss of 1.4% due to the local bit optimization, because only 2% of misses are local. `SWIM`, `MGRID`, and `GAFORT` show the largest benefit from this optimization.

In $CV = 2/LB$, the total system occupancy drops because the total drop in occupancy caused by sending fewer invalidations is greater than the marginal increase in occupancy introduces by longer handler occupancies. However, the two extensions $Hybrid/IC$ and $Hybrid/IR$ fail to demonstrate any significant performance advantages. $Hybrid/IR$ hurts performance in every benchmark except `WUPWISE`–where the technique only equals $CV = 2$.

In `SWIM`, the performance loss in the $Hybrid/IC$ extension is even greater than the $Hybrid/IR$ extension. Clustered invalidations only reduce contention when writes that have more than 16 sharers. This rare case has a large impact on `SWIM` because the arrival rate is higher than the other benchmarks, increasing its sensitivity to longer handler occupancies.

These results are significant because they demonstrate that occupancy introduces contention independent of where the occupancy is located. Naturally we expect hot spotting to occur at the home because maintaining a directory at one node in the system does introduce a centralized control point. However, attempting to reduce contention by shifting the occupancy burden to other nodes—which themselves must handle requesters for their local memory—only shifts where contention occurs. Longer handler occupancies increase the likelihood of contention.

5.3.4 The Occupancy Limit

To conclusively demonstrate that memory controller occupancy has a fundamental impact on overall performance, we examine the total occupied cycles across all six coherence protocols and extensions discussed to this point.

Table 5.3 summarizes the parallel efficiencies for all five coherence protocols across the SpecOMP2001 benchmarks. At 63 processors, there is a 42% difference in performance between the best and the worst protocols. This is surprising considering the minor functional differences between protocols. Boldface values in the table indicate the highest parallel efficiency protocol for each benchmark.

The total occupied cycles normalized to the basic coarse-vector protocol are presented in Table 5.4. The highest parallel efficiency protocol, coarse-vector with local bit, is also the lowest occupancy protocol, followed by hybrid, and hybrid with cluster invalidations.

Table 5.3: SpecOMP2001 Parallel Efficiency

Benchmark	*CV=2*	*CV=2/LB*	*hybrid*	*hybrid/IR*	*hybrid/CI*	*dynptr*	*best/worst*
SWIM	0.65	**0.96**	0.90	0.88	0.87	0.60	1.60
WUPWISE	0.75	**0.84**	0.79	0.77	0.77	0.54	1.55
MGRID	0.46	**0.55**	0.53	0.51	0.53	0.38	1.44
APSI	**0.71**	0.70	0.66	0.59	0.68	0.39	1.82
ART	0.48	**0.49**	0.42	0.37	0.39	0.14	3.50
EQUAKE	0.21	**0.22**	0.20	0.19	0.20	0.17	1.29
FMA3D	0.65	**0.71**	**0.71**	0.66	0.72	0.64	1.10
GAFORT	1.37	**1.64**	1.54	1.41	1.54	1.37	1.20
APPLU	0.80	**0.83**	0.63	0.59	0.64	0.52	1.59
Average	0.61	**0.68**	0.62	0.58	0.62	0.43	1.58

Table 5.4: SpecOMP2001 Total Occupied Cycles Normalized to CV=2

Benchmark	*CV=2*	*CV=2/LB*	*hybrid*	*hybrid/IR*	*hybrid/CI*	*dynptr*	*R(PE)*
SWIM	1	**0.44**	0.66	0.67	0.60	0.78	0.83
WUPWISE	1	**0.71**	0.97	0.97	0.93	1.44	0.98
MGRID	1	**0.71**	0.90	1.02	0.85	1.17	0.87
APSI	1	**0.98**	1.13	1.99	1.12	1.89	0.81
ART	1	**0.95**	1.04	1.20	1.05	2.59	0.96
EQUAKE	1	**0.71**	1.00	1.21	0.98	1.95	0.95
FMA3D	1	**0.61**	0.72	0.86	0.71	0.95	0.92
GAFORT	1	**0.77**	0.82	1.29	0.78	1.18	0.82
APPLU	1	**0.88**	1.03	1.57	1.02	1.67	0.81
Average	1	**0.74**	0.91	1.14	0.88	1.42	0.97

The hybrid with invalidations sent to the requester has 14% higher occupancy than coarse-vector and has the second worst overall parallel efficiency. Finally, the dynamic pointer protocol achieves the lowest parallel efficiency and the highest total occupancy. Boldface values in this table indicate the protocol with the fewest occupied cycles. The last column of Table 5.4 quantifies the correlation of overall occupancy with the execution times in Table 5.3. Two terms correlate if their correlation coefficient, R, is near 1. For our protocol study, the correlation of the protocol averages of 0.97 demonstrates the strong correlation between total occupancy and overall parallel efficiency.

Breaking down these two tables by benchmark, the relationship between total occupancy and performance holds. However, two notable exceptions are SWIM and EQUAKE.

SWIM achieves the best parallel efficiency with the lowest occupancy protocol, coarse-vector with local bit. However, the worst protocol is not the highest occupancy protocol, base coarse-vector, but dynamic pointer. Occupancy is critical for the total aggregate queuing delay present, but total occupancy is only a coarse measure. The high arrival rate of requests in SWIM and the sporadic arrival of cache miss requests—and replacement hints for each cache miss—contribute to the lower performance of dynamic pointer even with smaller occupancy than base coarse-vector.

EQUAKE has a small variation in parallel efficiency and a large variation in total occupancy. The low parallel efficiency suggests that there is little parallelism available in the benchmark. As such, large swings in total occupancy do not translate into large swings in parallel efficiency.

The remaining benchmarks track with the lowest occupancy protocol providing the highest parallel efficiency. Similarly, the highest occupancy protocol corresponds to the lowest parallel efficiency protocol. The extra occupancy does not arise from long execution times because each benchmark experiences roughly the same number of cache misses independent of protocol.

Correlation between high occupancy and low performance, even for protocols designed to minimize uncontended latency or shift occupancy to reduce the impact of hot-spots, indicates that protocols that keep total occupancy low by keeping handlers short and removing extra messages result in the best performance. The performance impact of queuing delay due to high occupancy is significant at 63 processors. $Hybrid/IR$ did not achieve better performance because the cost of extra occupancy negates the advantage of shorter remote point-to-point write latency and less contention at the home.

Figure 5.14 presents the geometric average execution time in seconds by occupied cycles at 63 processors. $CV = 2/LB$, the lowest occupancy protocol, demonstrates the highest performance because there are few extra invalidations and the occupancy of each handler is small. For the other protocols, there a limited zone where adding some occupancy generates little contention. The $Hybrid/IR$ protocol has about 57% more occupancy cycles than the $CV = 2/LB$ protocol at a cost of a 15% execution slowdown. Dynamic pointer, however, has crossed a knee at 63 processors where the cost of a little occupancy

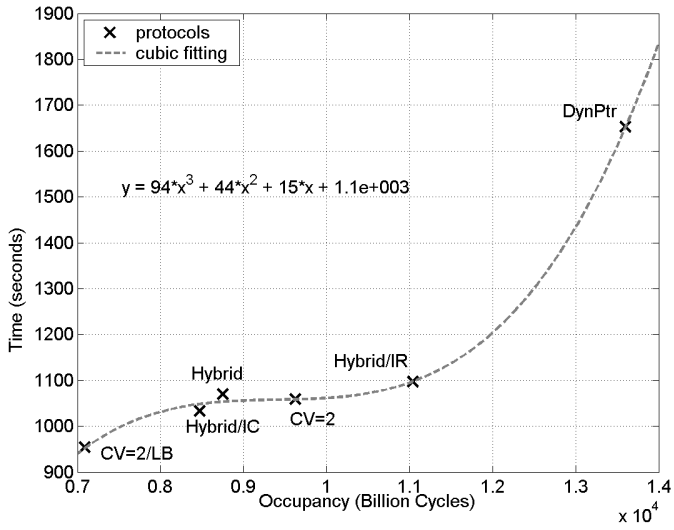


Figure 5.14: Execution time versus occupancy at 63 processors

becomes very expensive. This knee is called the *occupancy limit* because it represents the maximum level of occupancy for a given machine size that will still yield acceptable performance.

The occupancy limit is tied to the total number of single-threaded control points in the system. As processor count increases, the total potential locations for contention increases. More importantly, the total requesters for single-threaded control points increase. So the occupancy limit measured as a percentage of total execution time shrinks as processor counts increase.

What may be less clear is that occupancy limit holds true even with a faster memory controller and network. To demonstrate this, we drop the clock rate of the R10k processor to 75MHz to match the clock rate of MAGIC. Alternatively, one can view the system as one where instead of the memory controller being 3 times slower than the processor (as in the base system), it is now running at the same speed. This has the effect of decreasing the arrival rate of requests at MAGIC and lowering the occupancy of handlers relative to the R10k cycle time. Total occupancy does decrease: Coarse-vector by 23%, dynamic pointer allocation by 50%, and hybrid by 19%. $CV = 2/LB$ occupancy remains

the same. This makes sense—protocols that experience higher contention show a bigger drop in total occupancy. The relationship between the different protocols' execution times, however, remains the same. Dynamic pointer remains the worst protocol, and coarse-vector with local-bit remains the best. Contention effects are still important with faster memory controllers.

5.4 The Remote Access Cache Protocol Extension

In this section, we consider the addition of a 32MB remote access cache (RAC). It is an important addition to consider because caching remote data decreases point-to-point latency at the cost of occupancy. A RAC is also an orthogonal addition. Any of the coherence protocols considered in this paper could be extended to include a RAC. Forms of a RAC have been incorporated into systems like the Sequent STiNG [40] multiprocessor. Furthermore, a RAC affects only the trade-off between message arrival rates, remote latency, and occupancy but not the sharing set precision.

5.4.1 Latency Characteristics of a RAC

Like normal caches, a RAC reduces the observed latency by caching remote data locally. Memory requests to frequently accessed, exclusively used remote cache-lines benefit the most from this addition. The requester stores evicted cache lines locally and can access the data quickly if it uses the cache line again before the RAC evicts the cache line or the home intervenes. Directory headers that would represent the reserved portion of memory hold the RAC tags instead.

Adding any cache increases latency on misses. For RAC, the tag checks take time, regardless of a hit or miss in the RAC. MAGIC reserves a portion of main memory to store the RAC tags. Remote request handlers must read the tag state before sending data back to the local processor on a RAC hit or forwarding the request to the home on a RAC miss.

The uncontended latency overhead is approximately 200ns (45 processor cycles) or a 33% increase over the local memory access time of 600ns (135 processor cycles). Therefore, a RAC hit costs 800ns. A RAC miss also experiences a 200ns delay because of the tag check. The tag check overhead is reasonable compared with other studies that report remote access cache overheads of 21% to 36% of the local memory access time in their

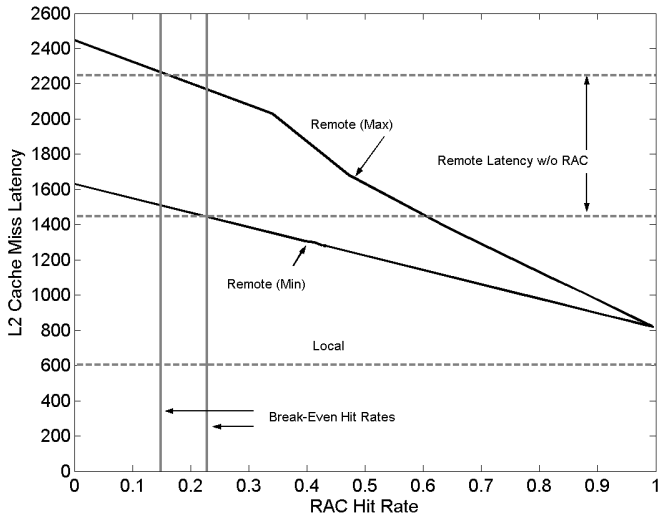


Figure 5.15: Remote L2 cache miss latencies versus RAC hit rates

designs [24, 45]. As a percentage, our RAC implementation has similar occupancy and latency characteristics to a hard-wired remote access cache.

On the hybrid protocol, a remote read miss may cost anywhere from approximately 1400ns to 2200ns depending on where the requester and home are located in the network. For the hybrid/RAC, the average L2 cache miss latency depends on the average RAC hit rate. Figure 5.15 illustrates the measured L2 cache miss latencies for remote requests as a function of the RAC hit rate. We present the best and worst case RAC hit latencies. The intersections of these lines with the hybrid protocol maximum and minimum remote L2 cache miss latencies indicate the *break-even* points. Somewhere between a 15% and 22.8% RAC hit rate, the average L2 cache latencies of the hybrid and the hybrid/RAC protocols match. We select the more pessimistic break-even point of 22.8% to place as much emphasis on performance loss due to uncontended latency as possible.

Prior work with RAC on a simulated 8-processor FLASH system [57] suggested high RAC hit rates in the range of 40% to 85% for a 16MB cache on the SPLASH-2 benchmarks. We use their RAC protocol code as a base for our 32MB hardware implementation. Figure 5.16 illustrates the per-thread minimum, average, and per-thread maximum RAC

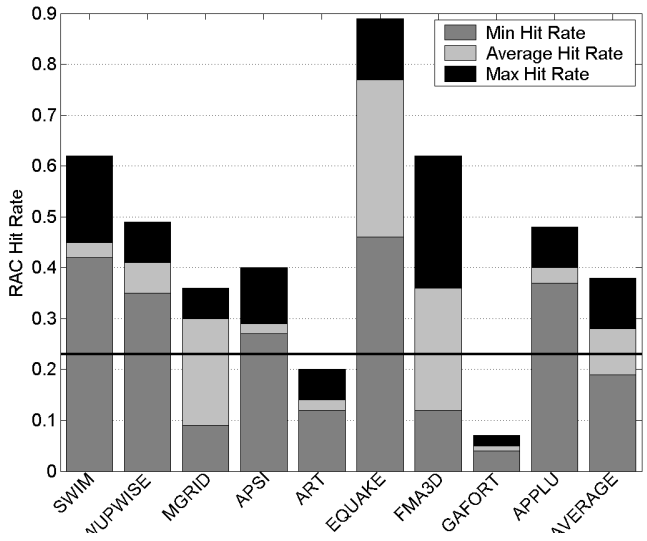


Figure 5.16: SpecOMP2001 RAC hit rates

hit rates for SpecOMP2001 benchmarks at 63 processors. Many benchmarks on average are higher than the 22.8% break-even threshold. The per-thread average hit rates for `ART` and `GAFORT` are well below the threshold. In addition, while `SWIM` and `WUPWISE` have minimum hit rates of 42% and 35% respectively, only a small fraction (5%) of their misses are remote. As a result, the performance effect, positive or negative, should be small. However, the hit rates for most benchmarks are well above the break-even point. Therefore, we expect those benchmarks to benefit from a RAC at 63 processors.

Given the measured RAC hit rates, we expect that extending the hybrid protocol to include a RAC would perform at least as well as the base hybrid protocol at 63 processors. Also, we expect that the `EQUAKE` benchmark would show significant benefit from a RAC. Unfortunately, the next section illustrates that occupancy significantly degrades the performance of the RAC, even for benchmarks that have naturally high remote access locality.

5.4.2 Occupancy Costs of a RAC

The total cost of the RAC protocol is greater than the uncontended latency of the tag check. MAGIC must store the tag state back to memory if it is modified and the home might send

more interventions to the local MAGIC to fetch and remove the data from the RAC. These extra interventions do not interrupt the processor's pipeline because the processor does not hold the data in its cache. However, the extra intervention handlers occupy MAGIC and cause contention with unrelated processor requests that access memory.

The RAC significantly reduces the memory bandwidth requirements of the memory system. As expected, the total network message counts drop from the hybrid to the hybrid/RAC protocol. As processor counts scale higher, the network bandwidth requirements decrease. The total messages sent on average across the network decreases by 24% at 8 processors and 33% at 63 processors. For the EQUAKE benchmark, the remote message counts drop by 47% and 70% at 8 and 63 processors respectively.

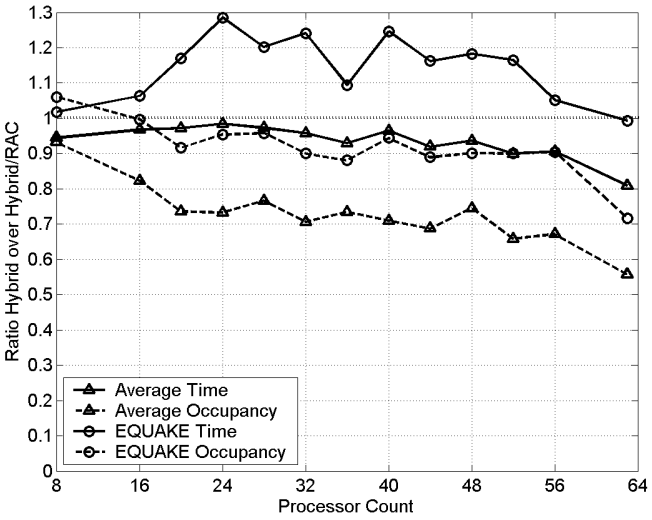


Figure 5.17: Key parameter ratios of hybrid over hybrid/RAC=32MB protocols

Figure 5.17 shows changes in execution time, occupancy, and RAC hit rates from 8 to 63 processors. The "Average Time" line illustrates the ratio of the average execution time of the hybrid protocol over the hybrid/RAC protocol execution time. We exclude ART and GAFORT from the average because their hit rates are not high enough to benefit from the RAC. The "EQUAKE Time" shows the ratio of the hybrid protocol execution time of the EQUAKE benchmark over the hybrid/RAC protocol execution time. Similarly, the

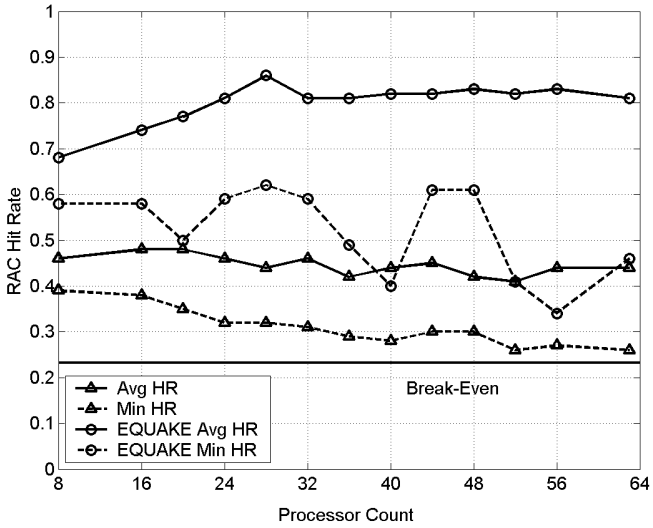


Figure 5.18: RAC hit rate versus processor count

occupancy lines illustrate the ratios of the total occupied cycles of the two protocols for the average benchmarks and EQUAKE. Figure 5.18 demonstrates that the poor performance does not result from poor remote access locality.

The RAC's performance is consistently poor—it degrades performance by 30% on average at 63 processors. While the EQUAKE benchmark does shown some performance improvement from 16 to 56 processors, it only breaks even at 63 processors. The performance of the RAC protocol mirrors changes in the total occupancy. The slopes of the average time and occupancy ratios are similar above 32 processors. The RAC protocol costs 66% more occupancy than the hybrid protocol at 63 processors.

These results expose a pitfall when choosing a benchmark suite to evaluate remote latency reduction techniques. Many architects choose a representative set of benchmarks like SPLASH-2 with a high percentage of remote cache misses to highlight the point-to-point latency reductions present in their proposals. However, high occupancy remote handlers can contend with the more common case of a high percentage of local cache misses. The SpecOMP2001 benchmark suite contains several benchmarks with a high frequency of local misses—indicative of optimized applications with natural locality.

Only 5% of ART's L2 cache misses access remote memory. However, the point-to-point latency penalty of a RAC miss does not account for all of the performance lost in this benchmark. Adding a RAC increases the execution time by 71% because the longer occupancies exacerbate a hot spot already present in the application.

This occupancy effect at larger processor counts is counter-intuitive given the wide adoption of the remote access cache in many modern designs. Clearly, the RAC protocol causes a reduction in both uncontended point-to-point latency and network bandwidth—especially for benchmarks like EQUAKE with high RAC hit rates. RAC is one example where a focus solely on latency and network bandwidth tells the wrong story. Extra occupancy of the handlers causes enough contention to effectively eliminate the RAC advantages at higher processor counts. Occupancy costs of a RAC would be smaller, although not absent, in a hard-wired approach. Nevertheless, these results indicate that the performance advantages of similar techniques that have been evaluated solely on the basis of latency reduction are potentially over-stated.

5.5 Quantifying Ideal Coherence Protocol Behavior

To understand the value of protocol optimizations further, we explore an analysis that assumes that all protocol overheads can be removed—leaving the true cost of communication due to access to shared-memory. The goal of this section is to determine the best-case performance gain an aggressive memory system design can provide. That is what is possible if the architecture community could solve all the negative latency effects present in these benchmarks and effectively manage the occupancy limit?

The Zero-Overhead Protocol

The first ideal memory system this section considers is one where there is no additional latency due to special coherence cases and no contention in the memory system. We call this a *zero-overhead* protocol. Such a system would have all of the benefits of a shared-memory model with latencies more typical of message passing architectures. Writes in this protocol are immediately served by a reply from the home before sending invalidations. The memory system would be designed so that occupancy would never introduce queuing

delay. This model assumes that legitimate contention for shared data could be removed from the program.

This model assumes "zero overhead" means with respect to the memory access latencies measured on FLASH. However, flexibility comes at a cost of both longer occupancies and latencies. For this ideal model, we ignore occupancy overheads. FLASH's uncontended local read latency requires 35 additional processor cycles compared to an SGI Origin 2000. The extra cycles represent a measure of the cost of flexibility. Some of the extra time arises from additional delay for MAGIC to schedule a handler, determine the directory entry address and read the directory state—all operations take longer than a hard-wired solution. In practice, the 35 cycles are small relative to the remote communication time of 300 to 900 cycles.

The All-Local Protocol

The second memory system has no remote communication and all misses were serviced with the same latency as a local miss. A symmetric memory system would have this property. However, scaling problems encountered in large-scale symmetric systems force the use of a distributed shared-memory architecture. A cc-NUMA machine could approximate this if the processor had perfect knowledge of future communication patterns and could aggressively prefetch data into its cache. The *All_Local* protocol assumes that any cache miss has the same latency as a local miss. For reference, we use the base local memory access time of 135 processor cycles. As processor clock rates have improved, this latency is now an aggressive target, but the specific values are not critical.

5.5.1 Methodology

The Zero-Overhead and All-Local protocols are ideal and do not correspond to any "real" protocol. This sub-section describes in more detail how to measure performance of these protocols. We use the $CV = 2/LB$ protocol as a base since it has the lowest occupancy of all the real protocols considered in Section 5.3. This analysis assumes no latency hiding in the base speedups—all removed latencies improve performance. Thus, the limit study presents the memory system improvements in the most attractive light possible.

$$T_{L2Miss} = T_m + T_w + T_{p2p} + T_c \tag{5.1}$$

Equation 5.1 partitions the total L2 miss stall time into measurable parameters. T_m is the minimum cache miss stall time for any cache miss received by MAGIC or 135 cycles. T_w measures the time that incoming requests are stalled while MAGIC runs another handler. T_{p2p} is the total point-to-point communication time to send and receive data from remote nodes. T_c is additional delay due to invalidation, interventions, and other coherence traffic including the remote communication required for local misses to invalidate remotely cached data. We include remote MAGIC processing time and queuing delay in T_c.

There are three common types of cache misses on cc-NUMA machines: *local* misses, *local requiring remote action (LRA)* misses, and *remote* misses. The key difference between a local miss and a LRA miss is the extra latency required to fetch data from a remote node's cache. An example of a LRA miss is an exclusive access request to a shared cache line requiring invalidations to remote caches. A fourth class of misses is a *remote requiring only local action (RLA)* miss; these misses are present in protocols that cache or migrate remote data (such as RAC). The base protocol does not include this case, but this limit study documents the maximum performance advantage that such a protocol would provide.

$$Speedup_{base} = \frac{8 * (Exec(8))}{(Exec(P))} \tag{5.2}$$

$$Speedup_{zero\_overhead} = \frac{8 * (Exec(8) - T_c(8))}{(Exec(P) - T_c(P))} \tag{5.3}$$

$$Speedup_{all\_local} = \frac{8 * (MaxCacheMiss(8) * (T_m(8) + T_w(8))}{(MaxCacheMiss(P) * (T_m(P) + T_w(P))} \tag{5.4}$$

To model the performance attainable by a perfect coherence protocol, the limit study quantifies an application's speedup as T_{p2p} and T_c approach zero. The zero-overhead protocol subtracts T_c from the base speedup (Equation 5.3). To model the uniform communication protocol that has perfect knowledge of communication patterns of the application, we remove T_{p2p} and T_c from the base execution time (Equation 5.4). We expect that as contention and communication are removed, the parallel efficiency of each benchmark will improve dramatically.

5.5.2 Quantifying Latency Terms

Every L2 cache miss creates a request requiring a response from MAGIC. In the case of a local read request, MAGIC runs a local `GET` request handler, which accesses memory and returns the appropriate data. This time is 135 processor cycles absent contention for MAGIC's handler processor and memory. Both local and remote requests share this overhead since they travel the same communication path to MAGIC and respond in identical ways once the data is fetched from the home's memory and loaded into a data buffer. We calculate T_m by multiplying the maximum cache miss count across all threads by 135 cycles. Note that while T_m is real—multiple memory access could overlap and reduce the impact of T_m on overall performance.

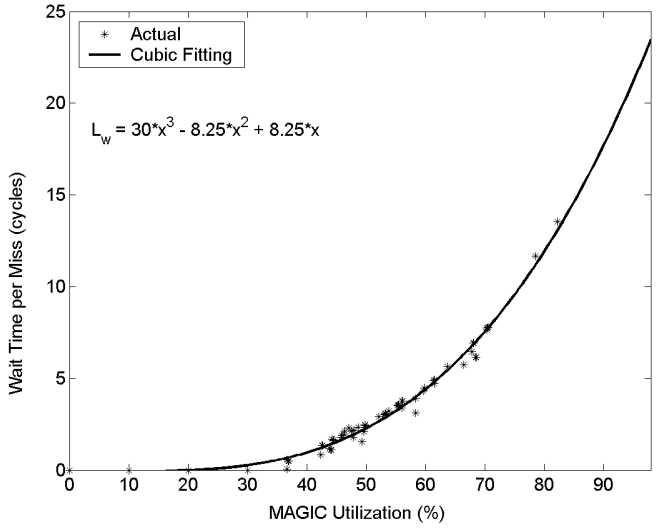


Figure 5.19: Local wait time, L_w versus MAGIC occupancy

Processor and network requests contend for the same MAGIC processor core. Incoming requests wait in a buffer for MAGIC to complete the currently executing handler. Figure 5.19 shows the average wait time per cache miss (L_w) as MAGIC utilization is increased. Because we have chosen a low-occupancy protocol, the SpecOMP2001 benchmarks' average MAGIC utilization is 15%—small enough that T_w is negligible. Significant hot spotting, however, can greatly increase T_w on a per-processor basis.

Each remote handler originating from the processor records the destination node of the request. To determine the point-to-point communication stall time, T_{p2p}, we multiply the distribution of remote misses by the unloaded remote point-to-point communication time for a particular source-destination pair. The maximum T_{p2p} for each thread indicates the total time saved by removing all point-to-point communication.

Measuring the contention and coherence overhead is usually difficult since the memory system on most cc-NUMA machines is opaque. A message buffer in the requester's MAGIC tracks the remote action latency. When a new remote request or local request requiring remote action arrives, MAGIC sends the remote request to the network and stores the cycle count in the message buffer. When the reply returns back to the requesting node, MAGIC forwards it to the processor and then rereads the cycle counter to calculate remote action latency. The `libmp` library enables this mechanism only during parallel sections of the benchmarks. The mechanism yields the total remote communication stall time, $T_{p2p} + T_c$, including extra coherence traffic for each thread.

5.5.3 Performance of the Ideal Protocols

Figure 5.20 presents the speedup of the Zero_Overhead and All_Local limit protocols over the normalized $CV = 2/LB$ protocol performance at 63 processors. Some benchmarks like ART and APPLU show dramatic speedups with an ideal protocol. Benchmarks that already have natural locality like SWIM and WUPWISE show no significant advantage as we might expect. Overall, the benchmarks improve achieve a speedup of 2X if one were able to remove *all* of the remote communication and contention.

The trend as processor counts increase from 8 to 63 processors demonstrates that the absolute performance advantages of these ideal protocols remain relatively constant. Figure 5.21 plots the average speedup for the Zero_Overhead and All_Local protocol from 8 to 63 processors. The Zero_Overhead protocol remains flat at a speedup of 30%. The All_Local protocol is erratic because the model makes many simplifying assumptions and some benchmarks vary widely on speedup. However, on average the protocol achieves absolute performance improvements from 70% at 8 processors to 100% at 63 processors. This makes sense since some applications have increasing communication at larger processor counts. More communication introduces more opportunities for improvement by reducing remote latency.

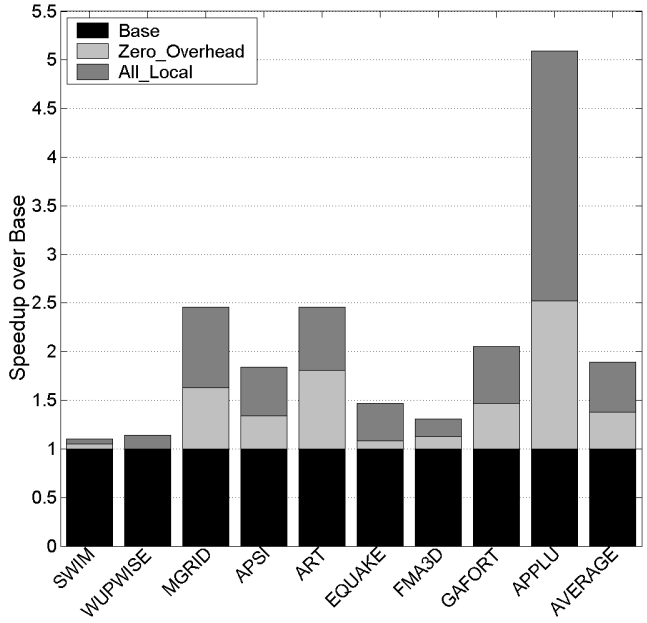


Figure 5.20: Ideal speedups versus benchmark

However, the ideal protocols are theoretical, and in practice the occupancy limit will reduce the performance gain shown in these graphs at larger processor counts. Reducing remote latency increases the memory system's sensitivity to occupancy because the arrival rate of requests for local memory would increase dramatically.

5.5.4 Scaling Impact of the Ideal Protocols

One key question is how these ideal protocols affect scaling at larger cc-NUMA machine sizes. This subsection discusses the speedup of the SpecOMP2001 benchmarks using our ideal protocol models.

Figure 5.22 shows the results of our coherence protocol experiment. On average, removing coherence traffic increases the SpecOMP2001 speedup at 63 processors (relative to 8p) from 63% to 73%. The figure clearly demonstrates that `EQUAKE`'s and `FMA3D`'s speedup does not drop due to the memory system. Their speedups remain flat when coherence and communication traffic is removed.

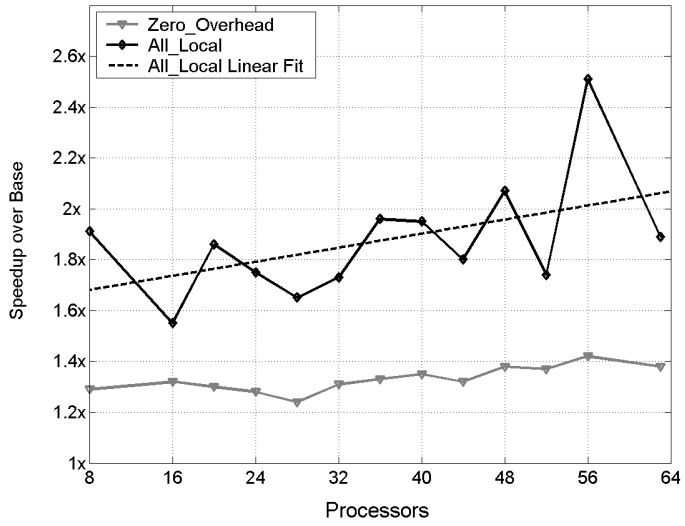


Figure 5.21: Ideal Speedup versus Processor Count

The most surprising result from Figure 5.22 is that removing coherence and communication delay *decreases* the average speedup to 42.84 because APPLU's and GAFORT's speedup drop. This is not a fault of the model but demonstrates super-linear speedup effects present in these applications, like cache and memory aggregation, that disproportionately impact performance at smaller processor counts. At higher-processor counts $T_c(P)$ and $T_{p2p}(P)$ are smaller than $T_c(8) * 8/P$ and $T_{p2p}(8) * 8/P$ respectively.

In particular, GAFORT experiences more hot spotting at 8 processors than at 63 processors, causing a drop in speedup when contention is removed. GAFORT's hottest loop is a shuffle of a large parent array between generations of a genetic algorithm calculation. The algorithm takes an element in the array and swaps it with another element ahead of it—locking both elements to ensure correctness. This causes a hot spot at the end of the array since more nodes are likely to swap with it. With more nodes, there is less contention for a particular MAGIC since each processor holds a smaller portion of the array. Removing coherence and contention traffic eliminates this super-linear hot-spotting effect causing a drop in $S_{zero\_overhead}$.

Figure 5.23 demonstrates how the latency increases from 8 to 63 processors. GAFORT

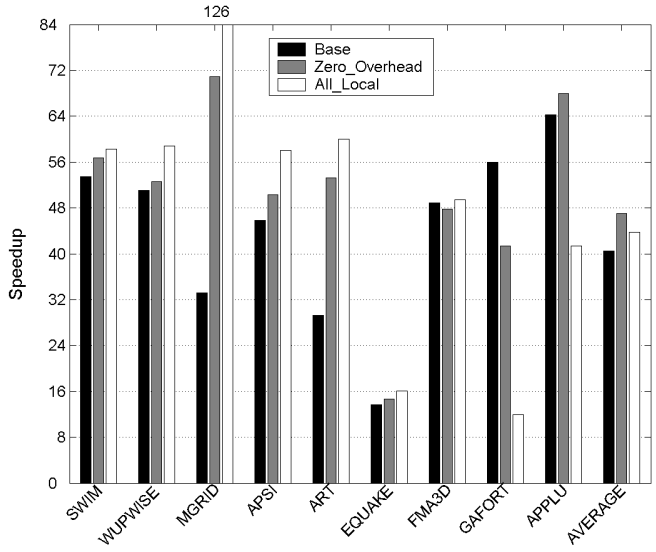


Figure 5.22: Ideal speedup at 63 processors

is the only benchmark where average latency drops from 8p to 63p and thus removing point-to-point communication eliminates this super-linear effect as well. By eliminating the impact of coherence and communication traffic, we expose the real source of dropping speedup—lock acquisition for elements toward the end of array.

APPLU's S_{all_local} drops for different reasons. There are fewer remote misses at 63 processors than at 8 processors due to cache aggregation. Several temporary arrays are misplaced causing unnecessary remote misses. Cache aggregation removes the frequency of remote misses at 63p. Removing communication traffic improves the parallel efficiency if the misplaced variables were originally placed correctly. In fact, when we fix the application in software so that the temporary variables are placed locally, the new speedup drops from near linear, meaning 63, to 33.39.

The speedup of MGRID increases by a factor of 1.94 when the ideal protocols remove remote coherence and communication latency. This benchmark experiences a rise in communication-to-computation ratio as processor counts scale. This effect increases the communication latency at higher processor counts as well as the coherence traffic to invalidate data shared along boundaries of array slices. When the ideal protocols remove

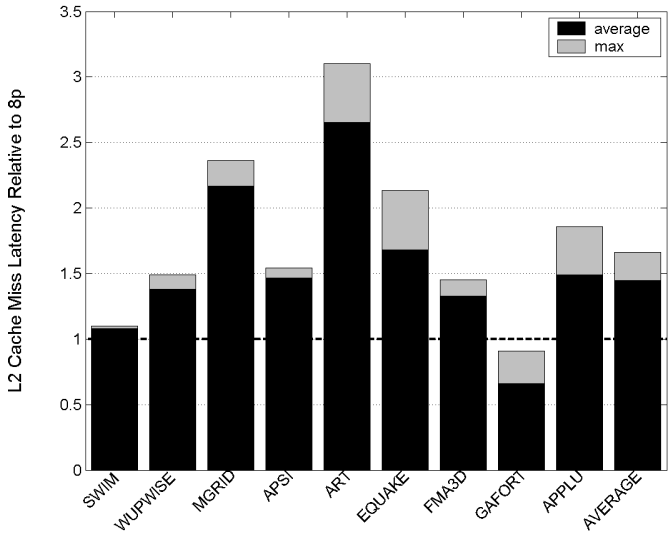


Figure 5.23: Latency Ratio - 63 processors over 8 processors

this communication and coherence latency, large-scale performance disproportionately increases because there is more communication to remove.

Despite making the most optimistic assumptions, memory system enhancements only improve speedup by 10% to 16% at 63 processors. While minimizing coherence and communication traffic can significantly increase the speedup of some benchmarks, other benchmarks show no gain since algorithm-scaling issues are exposed. T_w is a negligible term since the runs use the low-occupancy $CV = 2/LB$ protocol. However, contention of higher-occupancy protocols like dynamic pointer would increase T_w and other contention latencies and tend to eliminate relatively minor performance advantage.

5.5.5 The Impact of Latency Hiding

The analysis described thus far does not account for the impact of latency hiding. A processor may issue several requests at a time, the total cost of which may be partially hidden by latency tolerating mechanisms, including overlap in request handling. The impact of removing remote communication would be less if some latency tolerance that exists in the

application exploits this overlap. On the other hand, latency hiding could improve as serializing communication is removed, allowing more requests to be serviced in parallel. This section discusses the impact of latency hiding in more detail.

The ideal protocols are optimistic since they assume that improving latency translates 1:1 to a drop in execution time. Uniprocessor architects are acutely aware of the increasing memory gap and frequently employ techniques such as software pipelining, out-of-order instruction processing, and load/store buffering to reduce the impact of memory access latency. Multiple memory requests are often serviced in parallel since remote latency is typically much longer than what can be hid with one outstanding transaction. All of these latency hiding effects cannot accurately be measured on FLASH. However, the R10k processor can vary the number of outstanding L2 cache misses that the memory will service before stalling the processor from 1 to 4.

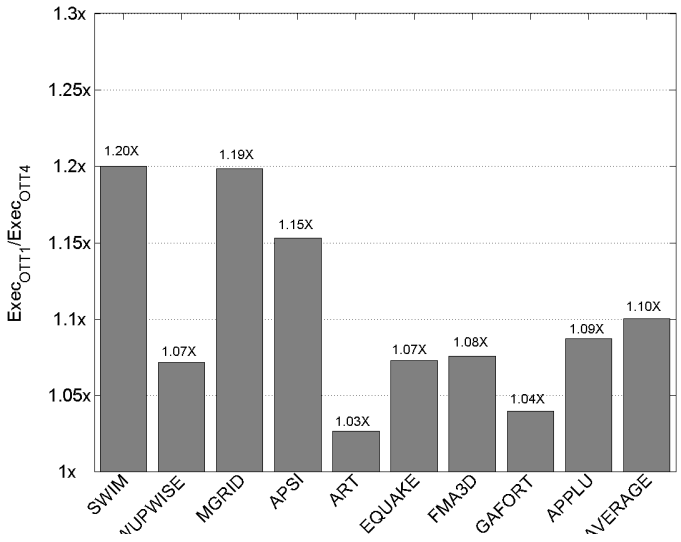


Figure 5.24: Latency hiding from 1 to 4 outstanding transactions

Figure 5.24 illustrates the percentage speed up by increasing the outstanding transaction tables per processor from 1 to 4 at 63 processors using the $CV = 2/LB$ protocol. Benchmarks speedup by 10% with 4 outstanding transactions, which indicates that the parallel efficiency and execution time improvements are narrower than this model measures.

Shrinking the size of the total outstanding transactions tables, or *OTT*s, has the effect of reducing load on the memory system and potentially lowering contention. However, the highest occupancy protocol, dynamic pointer, shows a speedup of 33% going from 1 to 4 outstanding transactions (not shown in Figure 5.24). By increasing the number of outstanding transactions, we increase the arrival rate of requests to the memory system. Therefore, high-occupancy protocols will naturally experience more contention. However, additional outstanding transactions allow overlaps in the queuing delays experienced by each, which improves the overall observed memory latency by the processor. This mitigation is limited by ILP and the memory system's capacity to the handle request bandwidth.

For example, the ratio of $CV = 2/LB$'s parallel efficiency to dynamic pointer's parallel efficiency improves from 1.89 at 1 OTT to 1.58 at 4 OTTs suggesting that the gap between low-occupancy and high-occupancy protocols may shrink as ILP increases. More likely, general-purpose applications lack sufficient ILP to benefit from additional OTTs. Complex protocols like dynamic pointer would only outperform simpler protocols if an application had an unusually high degree of data parallelism and no memory locality.

Some consider commercial applications to behave in precisely this way. For example, coherence protocol research at the University of Wisconsin [41, 42] examines commercial applications on symmetric multiprocessors. They note that memory performance is often dominated by three-hop cache-to-cache transfer misses. It is unclear whether the poor performance in their commercial applications arises from application-centric, architecture-specific, or machine-dependent communication. Unless the communication remains fundamental to the application's correct behavior, no exploitable spatial or temporal locality exists, and yet the application has ample data parallelism, our observation that simpler protocols will outperform more complicated ones will hold true.

Consider the overall execution time of the base, Zero_Overhead, and All_Shared protocol from 1 to 4 OTTs illustrated in Figure 5.25. In this figure each protocol executed with 1 OTT is divided by its equivalent with 4 OTTs. While there are some variations on average as processor counts scale, the speedup is relatively constant from 8 to 63 processors. This figure shows that latency hiding is likely to affect speedup only by a scalar factor for a single protocol.

Compare Figure 5.26 with its 4-OTT equivalent (Figure 5.21). These figures illustrate the speedup over the base protocol of the Zero_Overhead and All_Local protocols. The

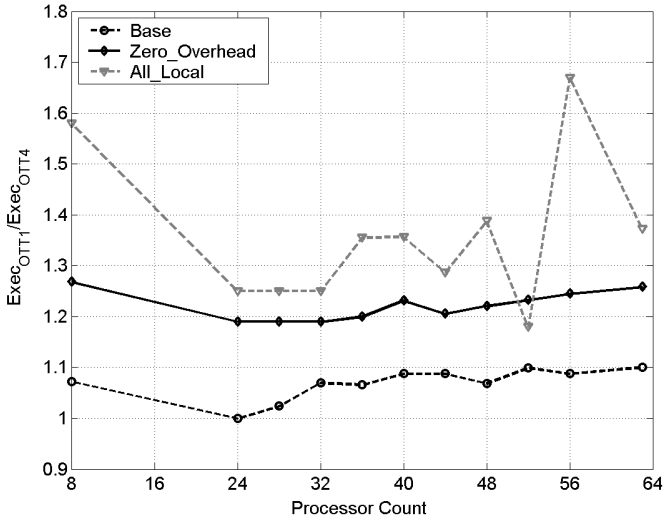


Figure 5.25: Execution time versus processor count with 1-OTT

original 4-OTT figure illustrates that the potential advantage of an All_Shared protocol is 1.7x to 2x depending on scale. However, with only 1 outstanding transaction the speedup is 1.4x to 1.6x. Additional latency hiding improves the overall speedup of the applications by approximately 20%

One might think that as the maximum number of OTTs increase, that the potential advantages would scale up beyond 20%. For example, perhaps with 8 OTTs the All_Shared protocol would improve the absolute performance by 2x to 2.4x. This is not the correct conclusion. It is the fault of the ideal model—which assumes a 1:1 correlation between latency reduction and performance. This assumption breaks down as the size of the OTTs increase because latency tolerance in memory system also increases and applications encounter ILP limitations. Shorter latency protocols show less benefit from latency hiding techniques.

An application will not see all of the benefits of a more aggressive memory system if it is not latency sensitive. Increasing the maximum OTT size makes an application less sensitive to latency, if there is sufficient ILP. Therefore, the base 4-OTT numbers used in the beginning of this section present a more optimistic picture of what a memory system

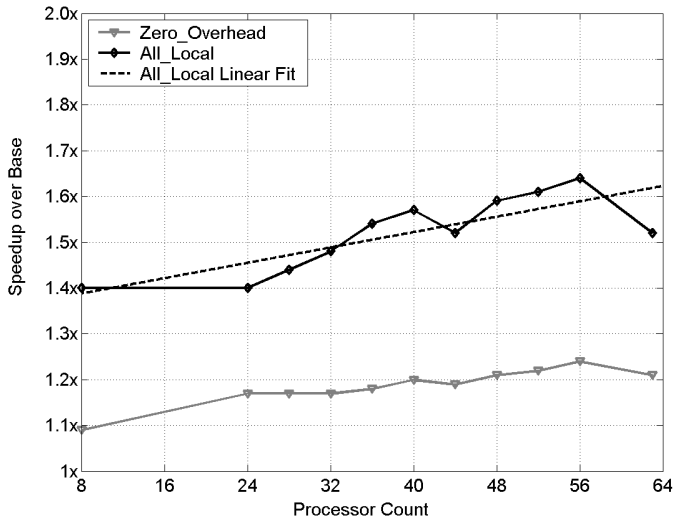


Figure 5.26: Ideal speedup versus processor count with 1-OTT

can do to remove memory latency. Dynamic pointer's occupancy costs have less impact because there are more opportunities for performance gains. Unfortunately, even this optimistic picture indicates the limits of what the memory system alone can do to improve poor performance.

5.6 Summary

This chapter examines the effects of contention, driven by protocol design, on the performance of multiprocessors intended to scale to larger processor counts. Although queuing delay due to contention at the memory controller is typically insignificant at smaller machine sizes with most memory controller implementations, such contention can be important with larger processor counts. This contention is significantly affected by memory controller occupancy, which in turn is dependent both on the node organization and on the coherence protocol.

Although simple protocols scale more effectively to large-scale machine sizes, they are less effective at solving structural problems in parallel programs such as hot-spotting,

false sharing, or excessive global communication, which some sophisticated protocols try to address. These results imply that prefetching and prediction will need to be extremely accurate to be effective. If either technique increases traffic beyond a minimum level, it is likely to lead to increased contention and little, or negative, performance improvement. In this study, the advantages of simplicity—lower occupancy and less contention—outweigh the gains from a more sophisticated protocol, at least for larger processor counts. This conclusion adds to the evidence that coherence protocols alone cannot solve parallel efficiency problems automatically.

Designing for low-occupancy protocols leads to simpler protocol state machines that limit messages and minimize occupancy. In an echo of the RISC arguments made 20 years ago about instruction set design, evidence from running real applications on FLASH points to the fact that simpler protocols scale best. Unfortunately, simple protocols are not likely to provide the same ease-of-programming advantages present on smaller-scale systems.

Furthermore, enhancing the memory system will likely provide only a scalar absolute performance advantage of at best 2x. These results do indicate that there are some benchmarks that show dramatic performance advantages from enhancing the memory system. These advantages are averaged out by other benchmarks that either show no change or a decrease in parallel efficiency with more aggressive memory system design. However, it is unlikely such advantages will realistically be captured on larger systems without hitting the occupancy limit, which drops as processor count grows and latency drops. A key question remains unanswered: how difficult is it to solve these communication bottlenecks in software? Are the problems difficult enough to solve that they require a hardware-only solution? These questions will be addressed in the following chapter.

Chapter 6

Software Bottlenecks and Optimizations

Given the pressure to keep the complexity of the memory system simple, this chapter gauges the difficulty of solving performance bottlenecks by tuning the application in software. The optimizations presented in this chapter reflect additional programming complexity required to achieve high performance. In some cases, the optimizations are quite machine-specific. Once understand, most are trivial to apply. On FLASH, discovering these bottlenecks was an easier process than on more traditional multiprocessors because of the ability to instrument memory system accesses. This chapter classifies bottlenecks into three categories: insufficient parallelism, excessive implicit communication, and synchronization and load-imbalance issues.

6.1 Increasing Communication-to-Computation Ratio

Increasing communication-to-computation ratio—set by the application and its choice of algorithm—is a key indicator of decreasing parallelism available at larger-scale. We measure this ratio by counting local and remote L2 cache misses seen by MAGIC during the parallel section of the code and track how this count scales from a uniprocessor to 63 processors.

Counting local and remote cache misses summed over all threads, only provides a coarse measure of communication changes. This count includes unnecessary communication due to false sharing, but it is useful for identifying potential benchmarks where necessary communication changes are affecting parallel efficiency. Figure 6.1 presents the

ratio of L2 cache misses on 63 processors over 8 processor runs of the benchmark. A ratio above 1 indicates that there is more communication at larger-processor runs.

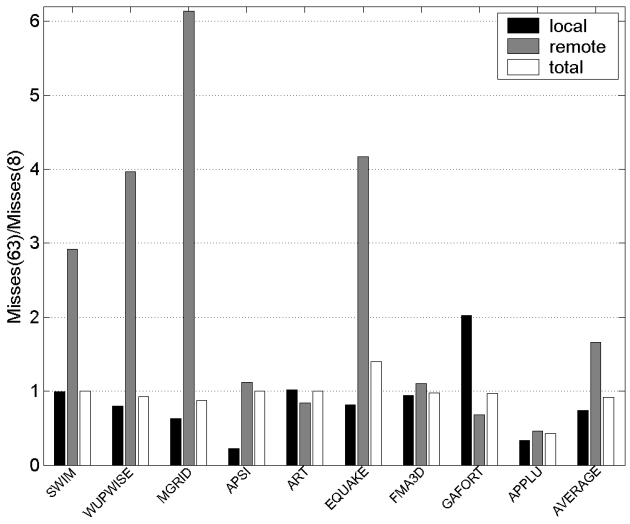


Figure 6.1: L2 cache misses relative to 8 processors at 63 proessors

The communication patterns in `MGRID`, `SWIM`, and `WUPWISE` are similar. Each thread iterates over a slice of a multi-dimensional array. Sharing occurs along the edges of the slices when threads read data placed on another processor. However, only in `MGRID` is the communication increase significant enough to lower the parallel efficiency considerably. The problem size is small enough that increasing remote misses by a factor of 6 significantly decreases parallel efficiency. Increasing the problem size by a factor of 8 increases the parallel efficiency from 50% to 95%.

`APPLU` has 50% fewer L2 cache misses at 63 processors than at 8 processors due to cache aggregation. For the remaining seven applications, their communication-to-computation ratios do not significantly change at larger processor counts. While `APPLU` achieves ideal parallel efficiency, it should perform *better* than ideal due to higher cache hit rates.

`EQUAKE` clearly experiences an increase in communication at 63 processors. This benchmark lacks sufficient parallelism to perform well. This benchmark's critical procedure calculates a sparse-matrix vector product. Figure 6.2 illustrates pseudo-code for

```
loop 1:
  for (j = 0; j < P; j++) {
#pragma omp parallel for private(i)
    for (i = 0; i < N; i++) {
      x[j][i] = 0;
    }
  }
loop 2:
  main_sparse_matrix_vector_product();
loop 3:
#pragma omp parallel for private(j)
  for (i = 0; i < N; i++) {
    for (j = 0; j < P; j++) {
      if (x[j][i])
        y[i] += z[j];
      }
    }
  }
```

Figure 6.2: Pseudo-code for EQUAKE's sparse-matrix vector product procedure

the sparse-matrix main loop. The array holds markers to indicate non-zero portions of the sparse-matrix. Loop 1 simply clears x. Loop 3 performs a type of reduction on y by collecting partial sums of y.

The number of loop iterations in loops 1 and 3 depends on the total number of threads, P. Therefore, the number of L2 cache misses for loops 1 and 3 increase linearly with total processor count causing the dramatic increase in communication-to-computation ratio observed in Figure 6.1.

If N is small, the performance of these loops would be insignificant. In this benchmark, N is one dimension of the sparse-matrix. Loop 2 does most of the algorithm work, but loops 1 and 3 eventually dominate performance. Loop 2's parallel efficiency is 41% from 8 to 63 processors. The parallel efficiencies for loops 1 and 3 are below 1% at 63 processors but the execution times for each loop are roughly equivalent. Other multiprocessors also experience poor parallel efficiency with this benchmark. On the SGI Origin 3800, EQUAKE only achieves a 30% parallel efficiency from 8 to 64 processors [1].

```
loop 1:
  /* -deleted- */
loop 2:
  main_sparse_matrix_vector_product();
loop 3:
#pragma omp parallel for private(j)
  for (j = 0; j < P; j++) {
     for (i = 0; i < N; i++) {
      if (x[j][i])
        y[i] += z[j];
      }
      x[j][i] = 0;
    }
  }
```

Figure 6.3: Optimized pseudo-code for optimized spares-matrix vector product procedure

For the SpecOMP2001-Large version of the benchmark, SPEC optimizes the benchmark to remove one of the loops. Figure 6.3 presents pseudo-code for the optimized benchmark. Functionally, the benchmark performs the same algorithm. However, they merge loop 1 into loop 3 and reorder the loops to take better advantage of cache locality. This simple change decreases absolute execution time by 66%. The scaling impact of this change is minimal. The speedup does improve from 21% to 27% on FLASH with the "large" binary using the same input set. However, the problem size is not large enough to provide decent parallel efficiency.

6.2 Unnecessary Implicit Communication

Of particular concern for cc-NUMA architectures is the degree to which unnecessary implicit communication decreases parallel efficiency. Extra communication is absent in other architectures that only allow explicit communication, but occurs on shared-memory machines due to cache-line false-sharing, page-level false-sharing, improper blocking, or poor data placement (as described in Chapter 2).

```
#define INTS_PER_CACHELINE 4

data_t x[INTS_PER_CACHELINE*P];
data_t y[INTS_PER_CACHELINE*P];
data_t z[INTS_PER_CACHELINE*P];
```

Figure 6.4: Pseudo-code for ART's variable declarations

Cache-line false sharing causes extra three-hop, cache-to-cache transfer misses that serialize threads and sharply increase cache miss latencies due to cache-line interventions. Three-hop misses can be avoided by restructuring the parallel section, by organizing data differently, or by reordering loops to take advantage of spatial locality.

Extra communication occurs when unrelated cache lines fall on the same page, causing hot-spotting at the page's home node even though there is no actual sharing for individual cache lines. Natural locality is lost since the OS organizes and places memory on a per-page basis. Cache-to-cache transfer misses do not occur in this type of false sharing since typically each cache line is used by only one node in the system. Page-level false sharing occurs in ART and APSI.

6.2.1 ART: Cache- and Page-Level False Sharing

ART is an image recognition benchmark. Each thread checks a small test image against a subset of a larger image to determine if they are identical. Each thread holds both images in its local memory. This benchmark makes a critical mistake when declaring per-thread statistics variables. Figure 6.4 presents pseudo-code for the declaration of these per-thread variables. Programmers use padding so that each thread has its own cache line to record matching information, but the benchmark hard-codes the number of integers per cache line to 4! The benchmark, therefore, incurs false sharing among threads on architectures with more than 4 integers per cache lines—in our case 8 processors share the same 128B cache line.

Furthermore, the cache lines all fall on the same page causing a large hot-spot on the memory controller on the home node. If the padded cache lines are distributed so that they also fall locally, occupancy on the home node's MAGIC falls from 97% to 62%. This

```
typedef struct {
   data_t x,y,z;
} ThreadPrivate_t;
ThreadPrivate_t *t;

main() {
#pragma omp parallel
{
  i = omp_get_thread_num();
  t[i] = memalign(PAGE_SIZE,sizeof(ThreadPrivate_t));
  bzero(T[i],sizeof(ThreadPrivate_t));
}
}
```

Figure 6.5: Optimized pseudo-code for ART's variable declarations

optimization, illustrated in Figure 6.5, groups per-thread variables together and places them on the CPU that executes the thread. Using this technique, parallel efficiency of the parallel section increases from 45% to 90%.

This benchmark exposes a critical hazard: smart programmers—fully aware of an application's communication patterns—can make simple mistakes that severely impact the scalability and portability of their code or fail to make their optimizations fully machine-independent.

6.2.2 APPLU: Poor Data Management

While the APPLU benchmark scales well to 63 processors, we expected super-linear speedup because of cache aggregation observed in Chapter 3. The cache aggregation arises from poor data management at smaller processor counts. Eventually, larger caches minimize the impact of this mistake. APPLU performs a sequence of Gaussian eliminations to perform a multi-dimensional LU factorization. While the structure and code are different, the algorithm shares many characteristics with the SPLASH-2 LU benchmark.

Our work with SPLASH-2 [22] found that the LU algorithm must be properly blocked to reduce contention and balance load. The APPLU benchmark practically implements the

```
data_t a[x,y], b[x,y];

/* initialization: */
#pragma omp parallel for
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
     a[x,y] = 0;
     b[x,y] = 0;
  }
}
/* use */
#pragma omp parallel for
for(block=0;block<B;block++) {
   i1,i2 = fx(blocks);
   j1,j2 = fy(blocks);
   for(i=i1;i<i2;i++) {
      for(j=j1;j<j2;j++) {
         func1(i,j); /* use of a buried in func1 */
         func2(i,j); /* use of b buried in func2 */
      }
   }
}
```

Figure 6.6: Pseudo-code for APPLU's scratch variable initialization and use

same high-level optimization.

The cache-aggregation arises from scratch variables used in the blocking procedure. Figure 6.6 illustrates a pseudo-code example of the key problem in APPLU. Variables a and b are initialized in a simple manner that breaks the parallel loop by the x dimension. The first N/x by y elements of the array will be placed on thread 0's node. However, the use of the variables—buried deep in subroutines called by func1 and func2—follow the blocked nature of the algorithm. Therefore, remote communication occurs because the initialization of the variables does not follow the use. At larger processor counts, the per-thread portion of a and b become small enough to cache, creating the super-linear effect.

As data set sizes increase, the sizes of `a` and `b` grow, limiting the caching of these variables and increasing the machine size required to remove the bottleneck through caching. The proper solution involves placing the variables consistently with their use. The initialization of the variables should follow the blocked nature of the algorithm. High-level procedures decide loop order and low-level and leaf subroutines use the scratch variables. Understanding the proper data initialization sequence is difficult given that the initialization and use are separated in the code.

Similar data layout optimizations could be applied to further speedup the `EQUAKE` benchmark. The program initializes scratch variables in order, but the input data instructs the program to adjust the loop order to balance load during the sparse-matrix vector calculation. The program can make better management decisions if the scratch variables are placed after loading the input set.

6.2.3 APSI: Poor Data and Cache Management and False-Sharing

Many effects like cache-line false-sharing, page-level false-sharing, and improper blocking, can interact in unusual ways. `APSI` experiences all three effects but from 8 to 63 processors has a reasonable parallel efficiency of 73%. Pages have not been placed with any consideration as to how they are used in the program. Only 3% of the total misses are local. The benchmark scales well because the uniprocessor and 8 processor runs also have high number of remote misses. In the SpecOMP2001-Large version of the application, some data placement issues have been fixed, but bottlenecks in the benchmark remain. Multi-dimensional arrays are not placed in memory consistently with their use. Most of the misses occur in three dimensional arrays ordered in `Z,Y,X` fashion, but loops stride through the arrays most often in `Y,X,Z` order. Blocking is not used to reduce the impact of misalignment so the benchmark experiences a large number of TLB misses (22.3 billion). Programs that experience a high number of TLB misses scale well as each TLB can be serviced independently [21]. However, the higher parallel efficiency belies the fact that there are serious blocking problems with the application. The SpecOMP2001-Large version of the benchmark places data more intelligently and identifies some reduction opportunities, but algorithm blocking problems remain.

Figure 6.7 shows the speedup curves for three versions of `APSI`. In the “opt1” version of the benchmark, we fix APSI so that data is placed consistently with data usage and loops

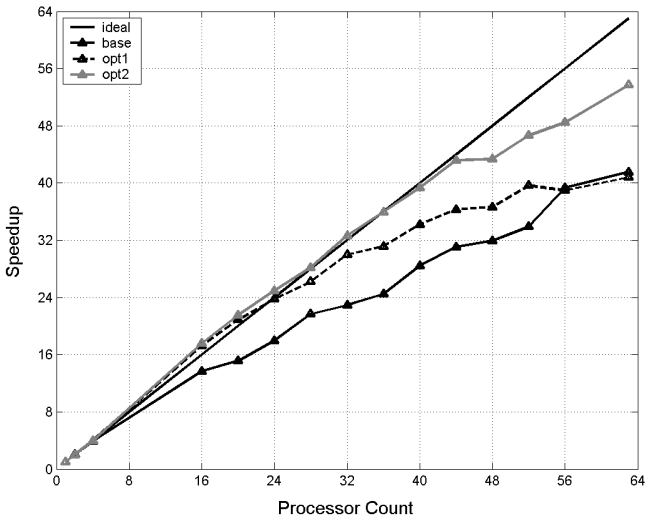


Figure 6.7: Speedup curves for base and optimized APSI benchmarks

are reordered to take advantage of cache locality and to reduce cache-to-cache transfer misses. We apply blocking to loops that experience large TLB misses, dropping the total number of TLB misses by 99% to 0.3 billion. The local cache miss percentage increases from 3% to 73%. Execution time improves by 21% at 63 processors. While the overall performance has improved considerably, the speedup curves remain nearly identical between the two versions above 48 processors.

Unidentified reduction variables account for the remaining parallel efficiency loss in "opt1". The "opt2" version removes the false sharing caused by the reduction variables. This optimization has also been applied to the SpecOMP2001-Large version of the benchmark where the reduction variables have been properly marked.

Developing the "opt1" version of the benchmark took approximately a week as it required a benchmark-wide change in the layout of memory. Removing the false sharing due to unidentified reduction variables, however, took only 15 minutes. This example shows that larger caches available at large-scale may mitigate the cost of remote point-to-point communication, but reductions and false sharing do not improve with scale.

6.3 Load Imbalance and Synchronization

The OpenMP standard provides pragmas for explicitly placing locks and barriers into application code. In addition, there are many instances where barriers are implied by the OpenMP pragmas. The `libmp` library places a barrier at the end of every parallel section to reduce confusion and programmer error due to race conditions between parallel sections. Locking occurs around OpenMP critical sections that can only be run by one thread at a time. Local synchronization is often required between threads that access shared data. We add timers in the `libmp` code to track when a thread first attempts to acquire a lock and finally succeeds. The thread that experiences the worst case lock acquisition time yields the total *locking overhead*. We measure load imbalance by calculating the difference between the first entry of a thread into a barrier, implied or explicit, and the last thread that arrives at the barrier. The load imbalance time includes global synchronization overhead required to implement the barrier.

Overall, small loops that cause frequent synchronization impact performance little. Varying barrier implementations from LL/SC to atomic operators on the memory controller (such as fetch-and-op) does not significantly change performance. Rather, load imbalance occurs due to structural problems in the algorithm.

6.3.1 GAFORT: Lock Contention

As mentioned in Section 5.5.4, `GAFORT`'s main bottleneck is lock contention during a shuffle operation in a genetic algorithm calculation. We now address what the software can do to improve the parallel efficiency.

Genetic algorithms are heuristics for finding local minimums of large problems. They are not precise exhaustive searches of a problem space. The shuffle operation used in `GAFORT` is a Fisher-Yates shuffle [33], where one element swaps with another element ahead of it. Load imbalance occurs when multiple threads attempt to lock the same element in the array. With many processors, each thread only has a small local portion, so most of its swaps will be remote.

We optimize `GAFORT` by relaxing the pseudo-randomness of the Fisher-Yates shuffle to reduce lock contention and remote communication. With a probability p, each thread performs the basic Fisher-Yates shuffle. With a probability $1 - p$, each thread performs

the Fisher-Yates shuffle only on its local elements. This has two advantages: scaling down remote communication as p drops and reducing the probability that different nodes contend for the same element.

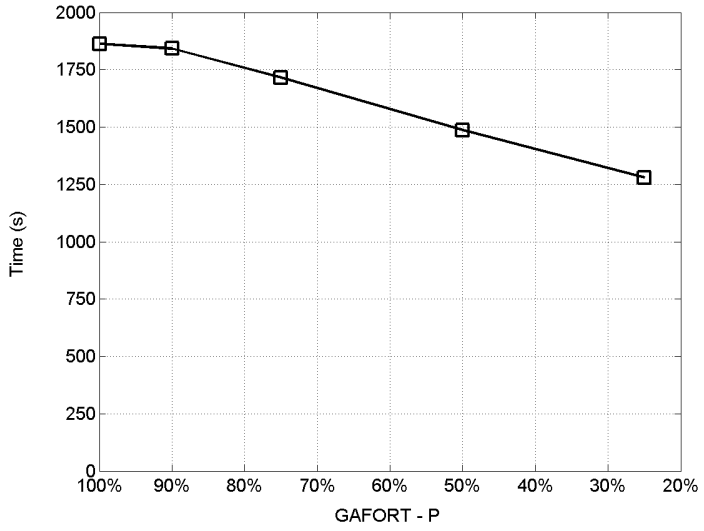


Figure 6.8: GAFORT time versus probability p

Figure 6.8 presents the execution data for our modified Fisher-Yates algorithm. We plot the x-axis in decreasing probability, since 100% represents the base performance. Initially, the load imbalance does not improve significantly until p drops below 90%. At $p = 25\%$ the benchmark still matches the expected output, but the parallel efficiency improves to 120%.

This optimization is ad-hoc, but it addresses two insights about the benchmark. First, the algorithm is a heuristic and thus the precise result does not depend on a perfect random shuffle. Second, load imbalance due to lock contention is the major bottleneck slowing the algorithm down. We make the choice to trade-off randomness for performance. A compiler would have a difficult, if not impossible, time doing this optimization unless the shuffle operation is abstracted out and replaced by the programmer with a library call. “Random enough” is an abstract and vague notion, and compilers are required to precisely and faithfully represent the code.

6.4 Summary

The penalty for programmer error or lack of optimization is high for large-scale multiprocessors. The benchmarks at 16 processors, a typical simulation-based machine size, achieve 85% parallel efficiency, so a programmer who uses small-scale runs to test and verify their code would miss many of these effects.

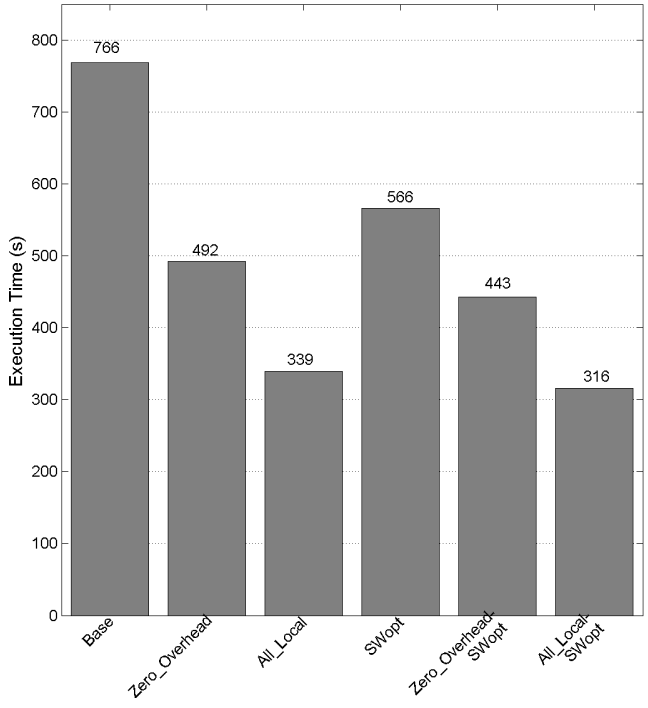


Figure 6.9: SpecOMP2001 execution time for untuned and optimized benchmarks with base and ideal protocols

Most of the optimizations applied to the SpecOMP2001 benchmark suite are basic. False sharing is a well-known problem but in several cases, simple mistakes that cause excessive false sharing have not been noticed, swamping parallel efficiency at large-scale. Loop reordering and blocking are common techniques for optimizing parallel algorithms.

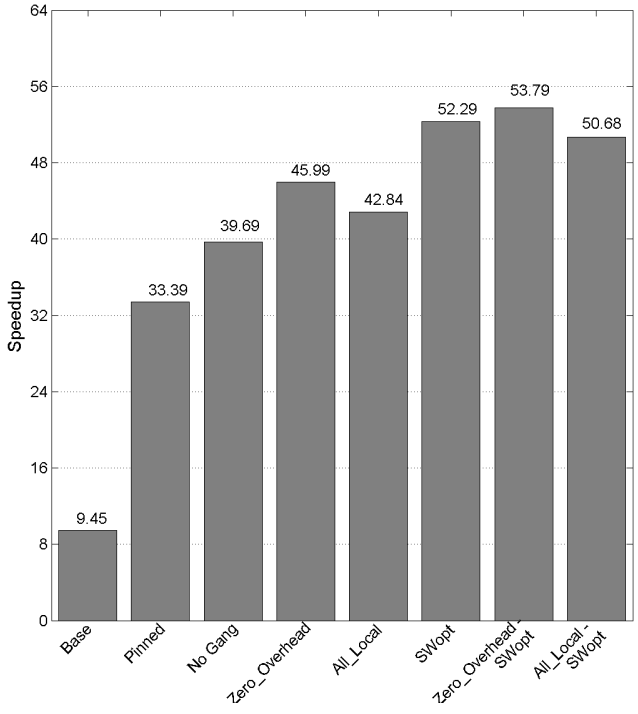


Figure 6.10: SpecOMP2001 speedups for untuned and optimized benchmarks with base and ideal protocols. Speedups for the ideal protocols are projected using Equations 5.3 and 5.4

In fact, tools like MemSpy [43] and FLASHPoint [21] already exist that identify appropriate program regions to block. It is harder, however, to apply these techniques on larger and more realistic benchmarks than on smaller applications and algorithm kernels.

Figure 6.9 graphs the absolute performance of the base SpecOMP2001 results with the derived coherence-free and communication-free performance numbers and the absolute performance of the optimized benchmarks. Repairing simple programmer errors recovered 75% of the performance gained by synthetically removing coherence traffic. While the ideal protocols do improve the absolute performance of the optimized applications, this improvement does not significantly affect the scalability of the applications.

Figure 6.10 shows the change in parallel efficiency with our software fixes relative to other hardware protocols and the operating system improvements. The software optimizations achieve a higher parallel efficiency than what is possible by hardware coherence

alone. Surprisingly, after applying the software optimizations, the ideal coherence protocols do not significantly improve the scalability of the applications.

It is, of course, unclear that an intelligent coherence protocol could eliminate coherence or communication traffic from a benchmark. Even if it did, the typically higher occupancies required to do so lessen the latency advantage at large-scale by introducing significant contention. The key obstacle to performance, therefore, is programmer's ability to identify, understand, and fix bottlenecks in software. Performance programming remains difficult. The programmer must still optimize their code for performance. This observation points to the need for the programmer and software tools to instrument and understand memory system behavior quickly—difficult to do in most cases because memory systems are opaque—to reduce the time required to apply optimizations in software.

Chapter 7

Conclusions

The FLASH project began nearly 10 years ago as an effort, among many, to study the impact of coherence protocols on the overall design of large-scale multiprocessors. The original designers also chose to design the system to compare the advantages of a shared-memory architecture's implicit communication model with a message passing architecture's explicit communication model. This dissertation in some respects draws that work to a close by addressing the questions raised by the initial FLASH group at the beginning of the project using the final real hardware version of the machine they proposed.

7.1 Conclusion

A big question was what types of coherence protocols would scale to larger machine sizes? Similar to the RISC revolution nearly 20 years ago, our results show that a simpler memory system is preferable to a more complex one. While "smart" coherence protocols do reduce the point-to-point latencies absent contention, the occupancy costs create significant queuing delay that reduces or eliminates the performance advantage of proposed changes at larger machine sizes. This is a negative result because much of the research in the shared memory community has focused on designing a more complicated coherence protocol. Furthermore, these protocols fail to remove problems introduced by unnecessary implicit communication or reduce the incremental tuning required by the programmer to generate high-performance, efficient parallel programs.

This work points to an emerging *programming gap* that limits the programmer from effectively using growing parallel resources. Hardware technology has improved dramatically since the first early multiprocessors like the Illiac IV. Today, we can construct multiprocessors with thousands of nodes at comparatively low cost. The emergence of low-cost Linux-based clusters, high-speed interconnection networks and multiprocessors on a chip point to the increased availability and lower cost of these larger scale multiprocessors. However, despite 40 years of research, it is still difficult to program high performance applications in a straightforward and machine-independent fashion. This difficulty persists in part because making programming mistakes is easy and the penalty for these mistakes scales with processor count.

Unless this programming gap is addressed by future researchers, larger multiprocessors will remain a niche market—useful only for a lucrative but small set of special-purpose applications. This observation should concern architects. The end of Moore's Law for uniprocessors has long been predicted. When the end arrives, architects will attempt to leverage parallel programming more aggressively to continue improving performance. The emergence of small-scale multiprocessor cores suggests that improvements to the uniprocessor provide diminishing returns already. However, unless this programming gap is addressed, parallel processing will only improve performance by a small amount for general-purpose computing!

Architects are keenly aware of the programming gap. Distributed shared-memory with cache coherence emerged as an approach to provide an interface to the programmer that focuses on critical communication. It is a "revolutionary" change over the traditional message passing approach where all communication must be made explicitly. Unfortunately, this dissertation presents a scaling flaw in the shared-memory model motivation: eventually even small communication hot spots become critical. With larger scale multiprocessors, the memory system matters less than the choice of parallel algorithm. Shared memory will not save the programmer from explicitly managing communication—especially at large-scale.

The most important contribution of the FLASH project is not its ability to mimic other memory systems—beyond simplicity the memory controller design or coherence protocol do not affect performance greatly—but the visibility it provides to the programmer to illuminate memory system behavior. This clarity allowed designers and programmers to quickly identify bottlenecks and fix them. Incremental tuning will always be necessary.

Rather than reducing how often one has to incrementally tune, architects should focus instead on reducing the time required to implement an optimization. This dissertation demonstrates that beyond simplicity, the memory system does not significantly improve performance or reduce the need for software tuning.

Machines of the future should therefore use resources in a conservative manner. Because larger processor counts are likely to use more resources less effectively, the programmer will find more benefit by using a smaller set of the total processors for new or untested code. After identifying standard parallel algorithms, the programmer will cede a large portion of his program to standard tools and libraries that make more effective use of parallel resources. Architects can reduce the tuning process by providing visibility to software tools so that the programmer can quickly identify key bottlenecks and implement appropriate machine-specific optimizations.

7.2 Future Work

Today, researchers continue to pursue new multiprocessor architectures. While our work provides a comprehensive analysis of the entire shared-memory multiprocessor system, we should not underestimate the ingenuity of future work that might address these problems in new ways. However, our work exposes some key pitfalls when analyzing and designing these large-scale machines. Future architects should reflect on these pitfalls encountered throughout this dissertation. More specific observations and lessons:

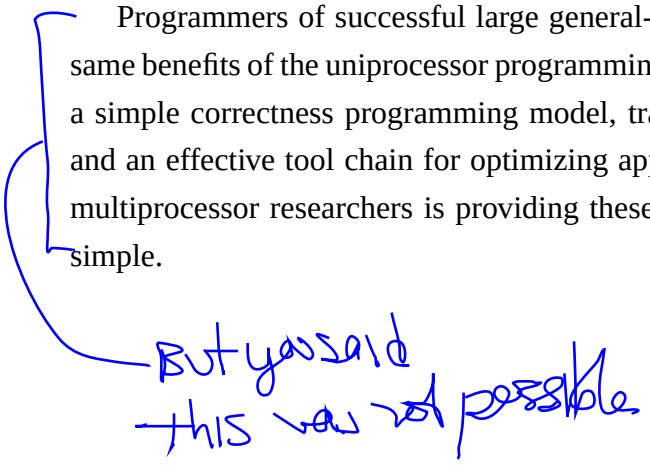
- Applications written well for shared memory should perform well on a shared memory multiprocessor. While applications with poor memory behavior present a more attractive target for aggressive memory systems, we must guarantee that any new technique does not penalize applications with natural memory locality. Therefore, evaluations of new memory system techiniques should include at least one application that is written well for their architecture model.

- Some applications will perform poorly even when executed with an aggressive memory system. Therefore, software technology that identifies and classifies performance bottlenecks are essential for tuning these applications. Hardware visibility designed

to assist these tools will cost some occupancy but will eventually provide a larger performance improvement than a pure hardware solution.

- Separating local and remote control paths reduces unnecessary contention. Local processors should always have priority access to local memory and never contend with remote requests that access unshared data.

Programmers of successful large general-purpose multiprocessors will experience the same benefits of the uniprocessor programming model: machine-independent performance, a simple correctness programming model, transparent view of where key bottlenecks lie and an effective tool chain for optimizing applications. The exciting challenge for future multiprocessor researchers is providing these benefits while keeping the memory system simple.



Appendix A

Interconnection Network Revisited

The original FLASH interconnection network topology [35] called for a hyper-cube mesh pattern similar to the Origin 2000. The original designers chose this topology because the components, mainly the CrayLink cables and the SGI SPIDER [19] routers, were identical in both machines. FLASH research focuses primarily on the design of the memory controller. Therefore, leveraging a commercially available network eliminated the need to design and test a high-speed network locally.

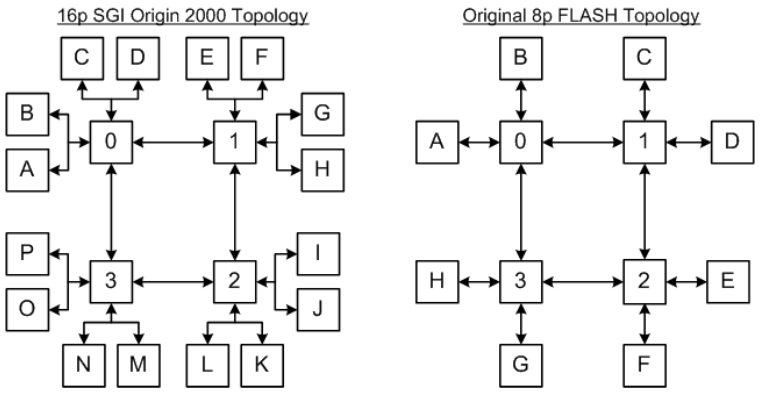


Figure A.1: 16-Processor SGI Origin and original FLASH interconnection network topology

Figure A.1 illustrates the topology of the SGI Origin 2000. Boxes labeled `A` through `O` represent processors. Boxes numbered `0` through `3` denote SGI SPIDER routers. Four processing nodes connect to each router across two CrayLink cables because each node

contains two physical processors. Each SPIDER router has 6 ports. The 16-processor Origin only requires 4 ports per router. More information describing the topology of larger SGI Origin 2000 systems can be found in [38].

The original 8-processor FLASH interconnection topology looks identical to the 16-processor SGI Origin 2000 system because a FLASH node contains only one processor. However, unlike the Origin, the original FLASH interconnection network topology deadlocks. This fact surprised the FLASH designers because larger Origin systems remain deadlock-free. An obvious question arises: how is the Origin able to maintain a deadlock-free network with the same interconnection network topology as the original FLASH machine?

The key assumption that hid deadlock in the FLASH network from the designers was that the use of virtual channels eliminated the need to manage physical network deadlock. Messages are sent between MAGICs on each node through the interconnection network. The interconnection network topology must be physically deadlock-free because the SPIDER routers do not allow messages to switch virtual channels when transmitting through the interconnection network.

Developing a deadlock-free coherence protocol requires the use of at least two independent networks [39]. The SPIDER router time-multiplexes four virtual channels [16] for each link to give the appearance of four independent networks for each processor without requiring the router to physically implement each network. MAGIC maintains separate hardware for each virtual network because it buffers incoming and outgoing requests for each virtual channel. Therefore, messages that arrive at any MAGIC terminate deadlock cycles. For example, MAGIC sends request and reply messages across separate virtual networks, so a node cannot deadlock with itself by sending requests to a remote node.

In the basic Origin topology a cycle occurs if processor `A` requests data from node `K`, processor `F` requests data from node `P`, processor `K` requests data from node `A` and processor `P` requests data from node `F`. If the messages are all routed through the network in a clockwise manner, a cycle occurs when all links `0` to `1`, `1` to `2`, `2` to `3`, and `3` to `0` block incoming requests. If the messages all route through the network in a counter-clockwise fashion, the cycle occurs in the opposite direction.

The original FLASH and the Origin topology can be made deadlock-free by redirecting routes to break cycles. Table A.1 shows deadlock and deadlock-free route tables on the

Table A.1: Routing Tables for 8-processor FLASH machine

Source	*Destination*	*Hop 1*	*Hop 2*	*Hop 3*
Deadlock Route Table				
A,B	E,F	0	1	2
C,D	G,H	1	2	3
E,F	A,B	2	3	0
G,H	C,D	3	0	1
Deadlock-Free Route Table				
A,B	E,F	0	1	2
C,D	G,H	1	2	3
E,F	A,B	2	3	0
G,H	C,D	3	2	1

original FLASH topology. In the deadlocked routing table, all routes travel clockwise around the routers. The fix requires re-routing messages from nodes G and H to nodes C and D in the counter-clockwise direction, increasing the bandwidth requirements of the link between routers 0 and 1. Processors E, F, G and H require the physical link between routers 2 and 3 to communicate with each other, but this link requires less bandwidth overall.

We assume that the Origin designers were aware of this network deadlock problem and made adjustments to the SPIDER routing tables to remove deadlock cycles. A 32 processor FLASH machine boots with the original topology once the equivalent routing table adjustments are made.

As processor counts increase, additional cycles emerge between new additional nodes requiring similar route adjustments. On Origin, under-loaded links can be used to break additional cycles, which balances each link's bandwidth requirements evenly. An Origin machine with more than 64 processor (32 nodes and 16 routers) organizes nodes into *metacubes* of 32 processors each in a bristled hyper-cube topology. Inter-metacube communication uses extra metacube routers to break additional cycles. Larger Origin machines require this topology switch because there are not enough routers to extend the basic hyper-cube topology to arbitrary sizes without introducing deadlock.

The FLASH machine lacks these inter-metacube routers due to project budget constraints that limited the amount of routers. Without these additional routers, building a FLASH machine with the Origin bristled hyper-cube topology is not possible without introducing network deadlock. However, we can build a 64-processor FLASH machine by

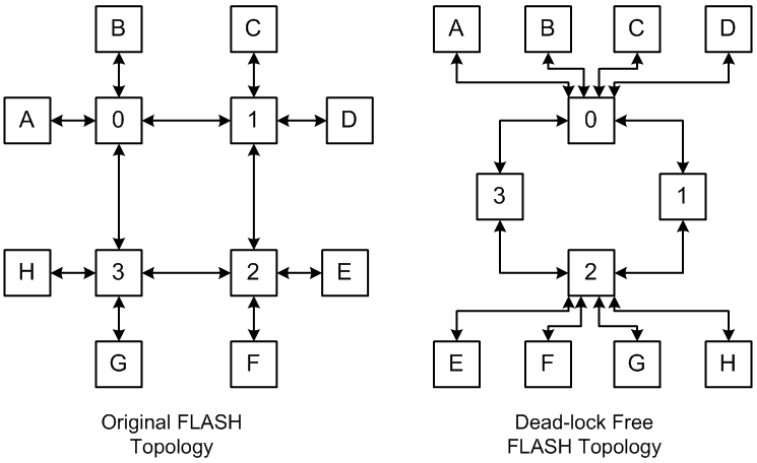


Figure A.2: 8-Processor FLASH interconnection network topology

switching to a double-mesh routing topology, illustrated in Figure A.2.

Four FLASH nodes connect to the same SPIDER router to form a cluster. This router is called the *cluster router*. The remaining 2 ports on the cluster router link the nodes with *mesh routers*. Routers `0` and `2` form clusters and routers `1` and `3` provide inter-cluster communication.

The new FLASH topology breaks cycles by maintaining separate request and reply routes. The deadlock-free topology appears to have an identical loop present in the original topology. However, internally the cluster routers do not route messages between mesh routers. Figure A.3 illustrates that internal cluster router connections do not physically link the mesh routers `1` and `3`. Therefore, messages that arrive from mesh router `1` remain independent of messages that leave through mesh router `3` and vice versa.

Figure A.4 illustrates a 16-processor FLASH metacube. Boxes labeled “N” represent a node comprised of the R10k processor, MAGIC and main memory. Boxes marked “C” represent cluster routers that connect 4 processors together. The “M” boxes denote mesh routers that connect clusters with other clusters on local or remote metacubes.

Figure A.5 illustrates how inter-metacube ports link together to form the 64-processor FLASH machine. The figure illustrates that bisection bandwidth between metacubes 0 and 1 is twice the bisection bandwidth between metacubes 0 and 2 or 0 and 3.

The FLASH topology shares some remote latency characteristics with newer clustered

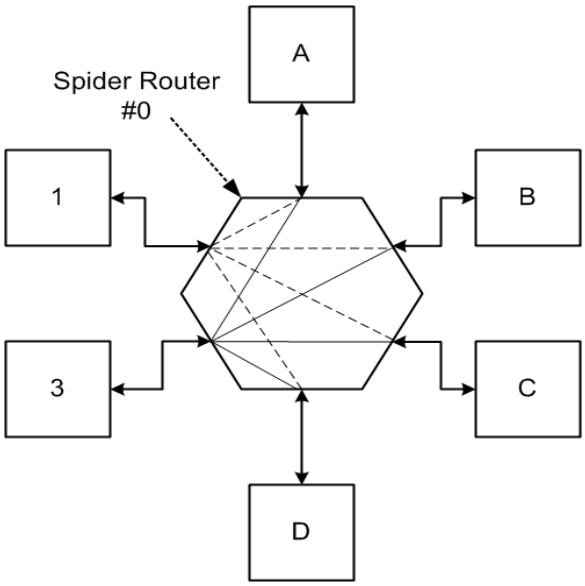


Figure A.3: Internal connections in FLASH cluster router

SMP systems, which have fast access to small-subset of nodes on the local SMP and longer latencies to remote SMPs. Local miss requests are always faster than remote requests. Remote requests that fall locally within the same cluster are approximately 2 times the local cache miss latency of 600ns. Remote requests on the local metacube are close to 3 times the local cache miss latency, and remote metacube requests cost almost 4 times the local cache miss latency. Therefore, remote requests latencies can improve remote request latencies by a factor of 2 if some remote requests are placed on a local cluster.

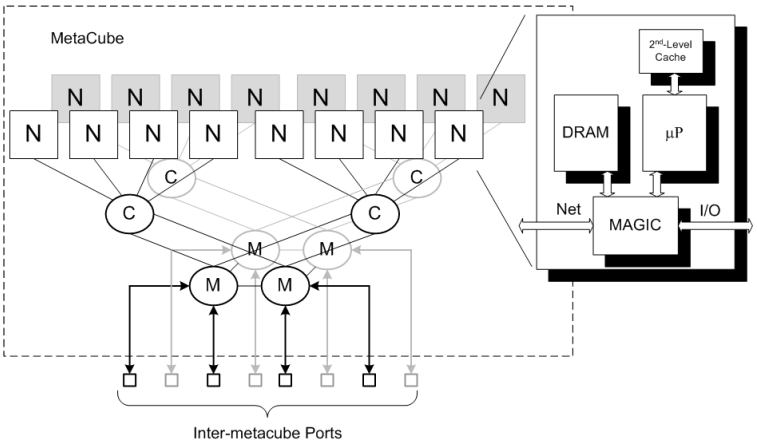


Figure A.4: 16-Processor FLASH metacube

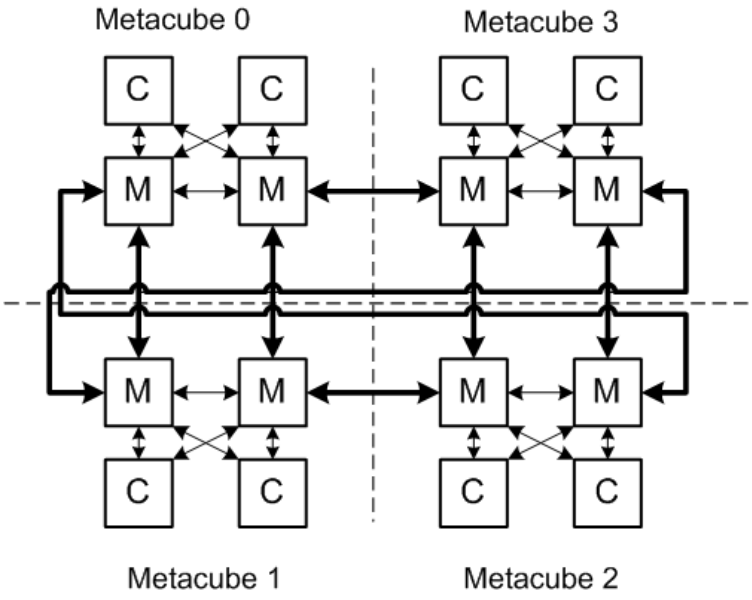


Figure A.5: 64-Processor FLASH interconnection topology

Bibliography

[1] SpecOMP reported results page. Website. `http://www.spec.org/hpg/omp/results/ompl2001.html`.

[2] G. Abandah and E. Davidson. Effects of Architectural and Technological Advances on the HP/Convex Exemplar's Memory and Communication Performance. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 318–329, June-July 1998.

[3] A. Agarwal, R. Bianchini, D. Chaiken, et al. The MIT Alewife Machine: Architecture and Performance. In *In Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, 1995.

[4] S. Amarasinghe, J. Anderson, M. Lam, and C.W. Tseng. An Overview of the SUIF Compiler for Scalable Parallel Machines. In *Proceedings of the Seventh SIAM Conference on parallel Processing for Scientific Computing*, February 1995.

[5] OpenMP Architecture Review Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Website, October 1997. `http://www.openmp.org`.

[6] ASC Q: Advanced Simulation and Computing Program. Website, 2004. `http://www.lanl.gov/asci/`.

[7] V. Aslot and R. Eigenmann. Performance Characteristics of the Spec OMP2001 Benchmarks. In *Proceedings of the European Workshop on OpenMP (EWOMP2001)*, 2001.

[8] V. Aslot et al. Specomp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science*, pages 1–10, July 2001.

[9] G.H. Barnes, R.M.Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes. The Illiac IV Computer. In *IEEE Transactions on Computers*, volume **C-17**(8), pages 746–757, August 1968.

[10] A. Birrell. An Introduction to Programming with Threads. *Digital SRC Research Report*, (35), January 1989.

[11] W. Blume et al. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Seventh Worskshop on Languages and Compilers for Parallel Computing*, pages 10.1–10.18, August 1994.

[12] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. *The Journal of Supercomputing*, 10(1):87–104, November 1994.

[13] L. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. In *IEEE Transactions on Computers C-27*, pages 1112–1118, December 1978.

[14] Chris Harris Cse. Wildfire: A scalable path for SMPs. In *HPCA*, pages 172–181, 1999.

[15] Z. Cvetanovic. Performance analysis of the alpha 21364-based hp gs1280 multiprocessor. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 218–229, June 2003.

[16] W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. In *IEEE Transactions on Computers*, volume **36**(5), pages 547–553, May 1987.

[17] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G. W. Stewart. Linpack users' guide. In *SIAM*, 1979.

[18] The Earth Simulator Center: Japan Marine Science and Technology Center. Website, 2004. `http://www.es.jamstec.go.jp/`.

[19] M. Galles et al. Spider: A High-speed Network Interconnect. In *IEEE Micro*, volume **17**(1), pages 34–39, January-February 1997.

[20] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[21] J. Gibson. *Memory Profiling on Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 2002.

[22] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[23] A. Gottlieb et al. The NYU Ultracomputer – Designing a MIMD, Shared-Memory Parallel Machine. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 27–42, April 1982.

[24] R. Grindley et al. The NUMAchine Multiprocessor. In *International Conference on Parallel Processing*, pages 487–496, 2000.

[25] A. Gupta, W.D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proceedings of the International Conference on Parallel Programming*, pages 312–321, August 1990.

[26] M. Heinrich. *The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols*. PhD thesis, Stanford University, 1998.

[27] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[28] J. Hennessy, M. Heinrich, and A. Gupta. Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):418–429, 1999.

[29] D. James, A. Laundrie, S. Gjessing, and G. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. In *IEEE Computer*, volume **23**(6), pages 74–77, 1990.

[30] J. P. Jones and B. Nitzberg. Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer-Verlag, 1999.

[31] S. Kapil. Gemini: A Power-efficient Chip Multi-Threaded (CMT) UltraSPARC Processor. In *Hot Chips 15*, August 2003.

[32] B. Kingsbury. The network queuing system, 1985. Sterling Software, Palo Alto.

[33] D. E. Knuth. In *The Art of Computer Programming, Volume 2, Third Edition*, page 145, 1997.

[34] P. Kongetiraer. A 32-way Multithreaded SPARC processor. In *Hot Chips 16*, August 2004.

[35] J. Kuskin. *The FLASH Multiprocessor: Designing a Flexible and Scalable System*. PhD thesis, Stanford University, 1997.

[36] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[37] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. In *IEEE Transactions on Computers*, volume **C-28**(9), pages 241–248, September 1979.

[38] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[39] D. Lenoski, J. Laudon, K. Gharachorloo, et al. The Stanford DASH Multiprocessor. In *IEEE Computer*, volume **25**(3), pages 63–79, March 1992.

[40] T. Lowett and R. Clapp. STiNG: A CCNUMA Compute System for the Commerical Marketplace. In *Proceedings of the 23nd Annual International Symposium on Computer Architecture*, pages 308–317, 1996.

[41] M. Martin, M. Hill, D. Sorin, and D. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 182–193, June 2003.

[42] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 182–193, June 2003.

[43] M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[44] E. Miya. Multiprocessor/Distributed Processing Bibliography. Website, April 2004. `http://liinwww.ira.uka.de/bibliography/Parallel/Eugene/index.html`.

[45] A.-T. Nguyen and J. Torrellas. Design Trade-Offs in High-Throughput Coherence Controllers. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[46] K. Olukotun et al. The Case for a Single-Chip Multiprocessor. October 1996.

[47] K. Otsuka and T. Watanabe. The Whole Earth Simulator: The World's Fastest Supercomputer. In *Hot Chips 15*, August 2003.

[48] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, October 1982.

[49] M. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *In the Proceedings of ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*, June 2003.

[50] U. Prestor. Snbench found online at. Website. `http://www.cs.utah.edu/~uros/snbench`.

[51] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[52] A. Saulsbury, T. Wilkinson, J. B. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, pages 276–285, January 1995.

[53] J.T. Schwartz. Ultracomputers. In *ACM Transactions on Programming Languages and Systems*, volume 2, pages 484–521, October 1980.

[54] S. Scott. Synchronization and Communication in the Cray T3-E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating System*, pages 26–36, 1996.

[55] R. Simoni. *Cache Coherence Directories For Scalable Multiprocessors*. PhD thesis, Stanford University, 1992.

[56] D. Sorin et al. Analytic Evaluation of Shared-Memory Systems with ILP Processors. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 380–391, June-July 1998.

[57] V. Soundararajan et al. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 342–355, June-July 1998.

[58] Linux @ Livermore: Thunder. Website, 2004. `http://www.llnl.gov/linux/thunder`.

[59] Top 500 Supercomputer Sites. Website, 2004. `http://www.top500.org/`.

[60] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[61] R. Wilson and other. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilters. In *ACM SIGPLAN Notices*, volume **29**(12), pages 31–37, December 1994.

[62] S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.