Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors

Vijayaraghavan Soundararajan

Technical Report: CSL-TR-99-789

November 1999

This work was supported in part by ARPA contract DABT63-94-C-0054.

Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors

Vijayaraghavan Soundararajan

Technical Report: CSL-TR-99-789
November 1999

Computer Systems Laboratory
Department of Electrical Engineering and Computer Science
Stanford University
William Gates Computer Science Building, 4A-410
Stanford, CA 94305-9040
<pubs@shasta.Stanford.EDU>

Abstract

Shared-memory multiprocessors are being used increasingly as compute servers. These systems enable efficient usage of computing resources through the aggregation and tight coupling of CPUs, memory, and I/O. One popular design for such machines is a bus-based architecture. However, as processors get faster, the shared bus becomes a bandwidth bottleneck. CC-NUMA (Cache-Coherent with Non-Uniform Memory Access time) machines remove this architectural limitation and provide a scalable shared-memory architecture. One significant characteristic of the CC-NUMA architecture is that the latency to access remote data is considerably larger than the latency to access local data. On such machines, good data locality can reduce memory stall time and is therefore critical for high performance.

In this thesis we study the various options available to system designers to transparently decrease the fraction of data misses serviced remotely. This work is done in the context of the Stanford FLASH multiprocessor. We utilize the programmability of the FLASH memory controller to explore a number of techniques for improving data locality: base cache-coherence (CC); a Remote Access Cache (RAC), in which a portion of local memory is used to cache remotely-allocated data at cache-line granularity; a Cache-Only Memory Architecture (COMA-F), in which all of local memory is used as a cache under hardware control; and OS-assisted page migration/replication (MigRep), in which the operating system migrates or replicates pages according to observed cache miss patterns. We then propose a novel hybrid scheme, MIGRAC, that combines the benefits of RAC and MigRep. We evaluate complete implementations of these schemes on the same platform using compute-server workloads (including OS effects), thereby providing a more consistent and detailed evaluation than has been done before.

We find that a simple RAC can improve performance significantly over CC (up to 64% gains). COMA-F improves locality but its additional complexity limits its gains versus CC (only 14% improvement). MigRep performs well (up to 33% gains) but does not handle fine-grain sharing as effectively as RAC or COMA-F. Finally, our MIGRAC approach performs well relative to RAC (up to 57% faster) and MigRep (up to 24% faster) and is robust.

Key Words and Phrases: COMA, data locality, CC-NUMA, page migration, page replication, unified architecture, cache coherence, MIGRAC, FLASH multiprocessor

Acknowledgments

This thesis would not have been possible without the gracious support of numerous individuals.

I would first like to thank my advisor, Professor Anoop Gupta, for his guidance throughout my graduate career. Anoop's seemingly boundless enthusiasm and ability to dissect problems quickly helped inspire me and push me to my limits. I owe him a great deal for helping me learn to do research. I would also like to thank my secondary advisor, Professor John Hennessy. Despite the pressures of being Computer Science Department Chair, then Dean of Engineering, and then Provost, John always willingly made time to give me feedback and advice. I would also like to thank Kunle Olukotun for serving on my reading committee, and for his friendship and guidance throughout my time at Stanford. Kunle's remarks improved the quality of this thesis, and his friendship helped ease my transition from MIT to Stanford.

Besides those faculty on the reading committee, I would like to acknowledge my MIT undergraduate thesis advisor, Professor Anant Agarwal, for introducing me to parallel processing and to research in computer architecture. I am also thankful to Professor Mark Horowitz, who served as the day-to-day leader of the FLASH project, and to Professor Mendel Rosenblum, who guided the SimOS team. I am also thankful to Professor Monica Lam for her friendship throughout my time at Stanford.

I benefitted a great deal from a number of very talented and friendly mentors while at Stanford. I am deeply grateful to Kourosh Gharachorloo, Rohit Chandra, Ben Verghese, Truman Joe, and Rich Simoni for helping me throughout my time at Stanford, both in school and outside school. I thank Alice Yu, Dan Scales, and Jennifer Anderson for advice on a variety of sometimes bizarre subjects, and for putting up with some of my more interesting musings. I thank Saman Amarasinghe, Martin Rinard, J.P. Singh, and Wolf-Dietrich Weber for helpful advice early on in my Stanford career. I thank Luiz Barroso, Steve Keckler, Ed Lee, Jens Skakkebaek, and Donald Yeung for helpful advice as I was preparing to leave Stanford.

I have had the pleasure of working with two groups of awesome system builders: the FLASH team and the SimOS team. I thank Joel Baxter, Jules Bergmann, Jeff Gibson, John Heinlein, Shankar Govindaraju, Mark Heinrich, Richard Ho, Hema Kapadia, Jeff Kuskin, David Ofelt, Dave Nakahira, and Jeff Solomon for developing the FLASH architecture and giving me tremendous amounts of help throughout this thesis. I am especially grateful to Jeff Kuskin and David Ofelt for guidance throughout the thesis-writing process. I gratefully acknowledge ARPA, which funded the FLASH project under contract DABT63-94-C-0054. I also thank Robert Bosch, Ed Bugnion, John Chapin, Scott Devine, Kinshuk Govil, Steve Herrod, John Huang, Dan Teodosiu, and Emmett Witchel for developing the SimOS simulator, and for helping me understand operating systems. I would particularly like to thank Kinshuk and Robert for answering my seemingly end-less questions as to why my simulations died, and for conspiring with Lucas Pereira to delay my graduation date through numerous challenging games of Foosball. I would also like to thank Kinshuk, Robert, and Lucas for teaming up with Scott Devine, Hoa Pham, and Chris Stolte to help improve my basketball skills.

I am thankful to a number of other colleagues who also helped make a lot of late nights in CIS and Gates more pleasant. I acknowledge Shigehiro Asano for numerous contributions in the early stages of this work. I thank Bharadwaj Amrutur, Andrew Chang, Bill Ellersick, Ricardo Gonzalez, Ziyad Hakura, Dave Heine, Rachid Helaihel, Victor Lam, Whay Sing Lee, David Lie, Dean Liu, Brian Murphy, Basem Nayfeh, John Owens, Beth Seamans, Kelly Shaw, Kanna Shimizu, Alan Swithenbank, Diane Tang, Tony Verma, Ken Wang, Ken Wilson, Steve Woo, and numerous others I may have inadvertently forgotten, for their friendship as I plodded along towards the Ph.D. I am particularly grateful to Ziyad and Kanna for being good office-mates and great friends. I also thank Navakanta Bhat, Ricardo Ramirez, Marc Schaub, Tim Waite, and Nazih Zard for their friendship throughout a tough first year and innumerable all-nighters at Stanford. I would also like to thank the members of the Stanford Tae Kwon Do club–in particular, Master Woo Kon Kim, Phil Acosta, Omri Beer, KC Chang, Chris Goldsmith, Chris Kim, Doug King, Vincent Lo, Everett Meyer, Dave Seniawski, and Sunay Tripathi–for giving me a great outlet for my thesis frustrations.

I am thankful to Charlie Orgish and Thoi Nguyen for keeping the machines in Gates in good running order, and for their friendship throughout my graduate school experience. Darlene Hadding, Margaret Rowland, and Shelley Russell were lifesavers in administrative matters. Shelley, in particular, was extremely helpful in a number of areas having nothing to do with Stanford, and I thank her for her friendship.

I would also like to thank a number of friends from MIT for their encouragement throughout my time at Stanford: Bret Harsham, Ye Gu, Lalit Jain, John Ma, Phylis Savari, Andrew Sheppard, Min Tung, and Angel Velez. I also gratefully acknowledge Eric Zylstra and Phil Liao for commiserating with me while they were enduring graduate studies at Berkeley. I thank Angela Chang and Beng-Hong Lim for a lot of good advice on personal and career matters over the years, and to Kate Nguyen and Lynsey Propeck for their encouragement as I was finishing up the thesis.

I would especially like to thank some very good friends, who put up with me throughout my graduate experience, despite my sometimes vocal dislike of hiking and camping: Luke Chang, Hooman Davoudiasl, Yan-ping Liao, Ulrich Stern, Donald Tanguay, and Rebecca Yang. I owe any improvements (however minor) in my bowling, interest in classical music, appreciation of spicy food, knowledge of German, guitar playing, and arguing ability to them. I consider them some of my closest friends, and I am extremely thankful for their support throughout my Ph.D. studies.

Finally, I would like to thank my family–my parents, my brother, and my two sisters–for their unfailing love and support. I owe them everything.

To my family

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Shared-memory multiprocessors offer the promise of high-performance computing by connecting nodes containing fast microprocessors and aggressive memory systems via a high-bandwidth, low-latency network. By working in a cooperative manner, this collection of processors can perform tasks in a fraction of the time that an individual processor would take.

Traditionally, multiprocessors have been used for running large scientific applications, for example, simulating the evolution of galaxies over time or the motion of airplane wings in a wind tunnel. Multiprocessors had been excluded from mainstream computing because they were expensive and difficult to build. The difficulty in writing software for such machines and excessive cost limited multiprocessor use to a few research labs where performance needs were critical.

Over the last few years, research in parallel processing has vastly improved the understanding of multiprocessor architecture, resulting not only in reduced design cycles, but also in multiprocessor-ready components. For example, state-of-the-art microprocessors now incorporate multiprocessor-ready features like support for external cache control and cache-coherence. Moreover, economies of scale have reduced the cost of components enough to make building multiprocessors practical. The resulting use of off-the-shelf components allows multiprocessors to be constructed using state-of-the-art components, providing concurrent computation without sacrificing the performance of uniprocessor applications.

Multiprocessors come in two main flavors: message-passing and shared-memory. In message-passing architectures, each node contains a private memory that other processors cannot access directly. Access to data located in a given node's private memory is accomplished by sending a message to that node, which accesses its memory on behalf of the requesting node. In contrast, shared-memory multiprocessors have the

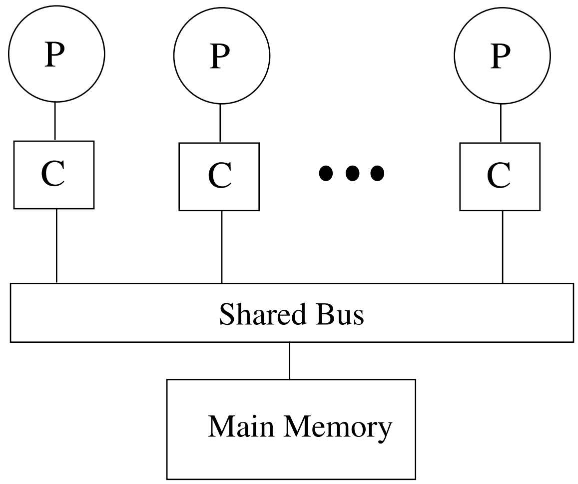


FIGURE 1. Small-scale shared memory multiprocessors. Each node consists of processor and cache, and the nodes are connected over a bus to the shared main memory.

notion of a single global address space. Processors are allowed to access any portion of the global shared memory directly.

In contrast to clusters of workstations connected by a message-passing substrate, shared-memory multiprocessors enable efficient usage of resources through the tight coupling of CPU, memory, and I/O. The global sharing provides the potential for scaled performance as resources are added: additional memory or processors can be utilized by any other processor in the system seamlessly, enhancing performance with low overhead. In addition, the administrative costs of maintaining a single multiprocessor can be much lower than that to maintain a cluster of workstations, increasing its commercial acceptance. As a result of their improved performance and commercial viability, shared-memory multiprocessors are being used increasingly as compute servers: for example, they are utilized for software development, CAD tools, web servers, and database engines.

In small-scale shared-memory designs, the processing elements are connected over a bus to the shared memory, as depicted in Figure 1. These bus-based designs are referred to as symmetric multiprocessors, or SMPs, because the nodes are symmetric to one another. In particular, the processors are equidistant from the main memory, and the latency to access memory is the same for each processor[1].

The SMP design works well with a small number of processors. However, as the number of processors increases, the bus can become a significant bottleneck as the processors attempt to access the single, centralized memory. For large-scale, commercial multiprocessors, a more scalable design is desired. The typical design for scalable multiprocessors, shown in Figure 2, involves distributing memory throughout the nodes of the machine and using a scalable interconnection network, rather than a shared bus. Distributing memory

1. These machines are also called UMA, or Uniform Memory Access latency, machines.

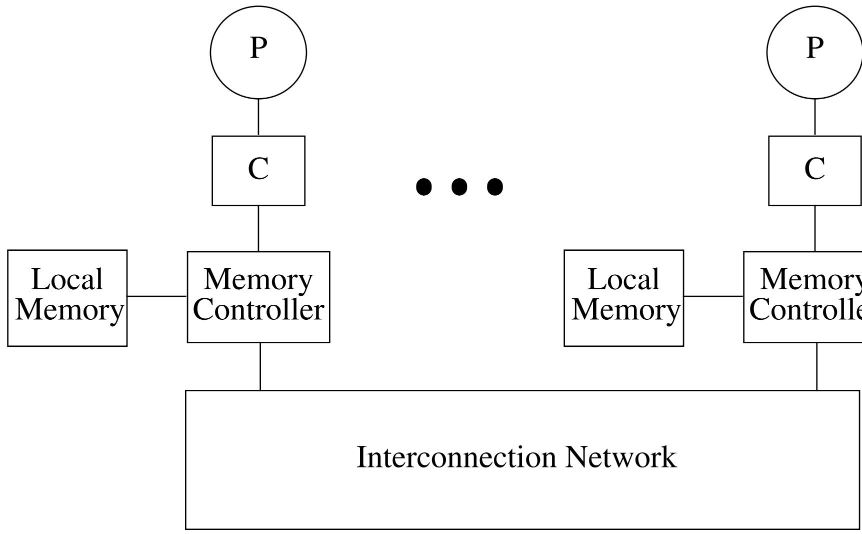


FIGURE 2. Architecture of distributed-memory multiprocessors. Each node contains a CPU, its associated cache, a portion of the globally-distributed main memory, and a memory controller. The nodes are connected by a scalable interconnection network**.**

throughout the machine prevents accesses to disjoint portions of the memory space from being serialized, as would happen in an SMP. Utilizing a high-performance interconnect like a hypercube or mesh allows more processors to be added to a system without sacrificing communication bandwidth, unlike in a bus-based design, in which each additional processor takes away bandwidth from other processors.

One drawback of distributed shared-memory multiprocessors, or DSMs, concerns the distribution of memory resources. Because memory is distributed, a desired datum may reside far away from the processor that needs it. On a cache miss, time spent retrieving such a datum increases the total execution time of the workload, hurting performance. Reducing the memory stall time component of execution time thus becomes important in improving application performance.

One method for reducing the memory stall time of applications is to improve data locality–that is, to try to increase the probability that a desired datum resides on the same node as the requesting processor, rather than on a remote node. Programmer-assisted techniques like modifying an application to statically place data near where it will be used or inserting prefetches of desired data can help. However, these methods are useful primarily for applications in which access patterns are predetermined or predictable, or for applications in which the source code is available. Compute-server applications like databases, however, are more dynamic than scientific applications: the data access patterns are unpredictable, and the operating system moves processes between processors to maintain load balance. As a result, static data placement techniques do not work, and it is also difficult for the programmer to insert prefetch instructions. If the programmer does not have the source code, then neither technique is even possible. We desire more general locality-enhancement schemes that work across different types of applications and do not require programmer intervention.

Caches are one general mechanism for achieving better data locality: these small, fast memories close to the processor are responsible for storing copies of data both from remote memories as well as the local memory. When a datum is initially accessed, it is stored in the cache, and subsequent accesses are satisfied by the cache, instead of requiring costly remote accesses. There is a practical limit to cache size, however, bounded by the processor clock rate requirements and on-chip or board space requirements. This limit means that caches cannot be arbitrarily large. As a result, applications with large data sets can easily overflow these caches, eliminating the locality improvements of caching. In multiprocessors, overflowing the cache can be far more costly than in a uniprocessor, since the latencies for accessing data in remote memories can be far greater than the latency to access local data. For example, in some machines, remote latencies are ten times higher than local latencies[LoC96]. In order to accommodate such data-intensive workloads on multiprocessors, other methods of improving data locality are needed.

1.1 Research Goal

This thesis explores a number of techniques for improving locality in compute-server workloads on distributed memory multiprocessors. The focus is on compute-server workloads, because of their dominance over scientific workloads in the marketplace, and because of the relative lack of research for data locality improvements on such workloads. Compute-server workloads are particularly tough to deal with because the dynamic access patterns make standard techniques such as prefetching or static data placement difficult. In addition, the source code for applications like compilers or Verilog simulators is often not available, so such optimizations are not even possible. Instead, more general techniques similar to caching are desirable.

The key to improving data locality is providing a flexible mechanism for allowing data movement. Ideally, it would be possible to change the amount of space available for replication, and to do so without undue complication. If possible, such a mechanism should be transparent to the programmer, and robust in the face of changing access patterns. Many researchers have proposed extending the concepts of caching to achieve such goals. For example, consider the conventional design of a shared-memory multiprocessor. Each node contains a CPU with its associated cache and a portion of the globally-distributed main memory. Each CPU can access data allocated on any other node. Both remotely-allocated data and locally-allocated data are copied (*replicated*) into the cache when referenced. The local and remote memories are used merely for *backing store*–that is, when the datum is evicted from the cache, it is written back to the local or remote memory from which it was copied. The local or remote memory that is the backing store is the memory from which the datum is allocated, and is often termed the *home* node.

There is no intrinsic reason that replication need only occur in the caches. Main memory is significantly larger than cache: for example, the SGI Origin 2000 multiprocessor contains space for up to 4 GB of memory per node, but only a maximum 8 MB of cache per node [LaL97] (each node contains two processors, each of which can have up to 4 MB of cache). Suppose a portion of memory were used as a cache as well, to supplement the processor caches: when a datum is referenced, it is copied not just into the processor cache, but also into the portion of memory used for caching. Although the processor cache is relatively small and can easily overflow, data evicted from the processor cache can be written back to the local “memory cache” instead of back to the remote memory from which the line is allocated. Because the memory cache can be much larger than the processor cache, evictions from the memory cache will be much less frequent than evictions from the processor cache. In the case of a remotely-allocated line, subsequent references to the datum, even if they miss in the processor cache, can be satisfied in the local memory cache, without requiring costly network accesses. Given that the ratio of remote to local latency can be high, saving remote reference latency can be quite a win. Even if only a modest portion of memory were used as a memory cache, the effective cache size per node can be dramatically increased: if only 10% of the available memory per node on the Origin 2000 were available for caching, then the effective cache size on the node would be 400MB, much larger than the default of 8MB.

Another option for caching is to perform data *migration*: if a datum is initially allocated in some remote memory, but is largely accessed by another node, then it would be helpful if the datum were moved (*migrated*) so that it is allocated out of the requesting node. In this situation, if a migrated datum is evicted from the requestor’s processor cache, then it is now written back locally, since there is now a locally-allocated backing store for the line. In the absence of migration, the datum would have had to be written back to its remote home, and the requestor would have to incur a network access to write it back, and then to retrieve the datum if requesting it again. One distinction between replication and migration is that in replication, the home node stays fixed, so if a line is evicted from the replicated node, it is still written back to the home. In migration, the home moves to the requestor’s node. Another distinction is that in replication, memory resources are allocated on multiple nodes for a given data item: one copy is kept at the home, and other copies are kept at the requestor nodes. This can result in wasted memory, especially if multiple nodes keep copies of a datum that they rarely access. In contrast, migration does not increase the amount of memory used in the system: the single copy is moved throughout the system as it is needed.

Depending on the approach, utilizing caching in memory can be very flexible. First, the amount of memory used for caching can be varied, either manually by rebooting with a different size setting for the memory cache, or automatically by the operating system. In addition, the rate at which data is replicated can be varied by only replicating when certain conditions are met. By selective replication of lines only when the application is likely to benefit from such replication, robustness can be achieved.

As the preceding discussion indicates, the design space for replication and migration is quite large. The caching can be under hardware (memory controller) control, under software (operating system) control, or even under a combination of hardware and software. The caching can be *eager*, in which a datum is replicated/migrated as soon as it is referenced, or *delayed*, in which it is referenced only after repeated misses, and only under certain conditions. The caching can be done at a cache-line granularity, at a page granularity, or even at a program-level object granularity. Some important examples that have been proposed include using memory as a Remote Access Cache (RAC), using all of memory as a cache (COMA), or using the operating system to migrate and replicate pages into local memory (MigRep).

RAC. A number of commercial and academic machines have used main memory as a cache only for remotely-allocated data [LLJ+92][Dat97][LoC96][RLW94][BrA97], also known as a *Remote Access Cache*, or RAC. A datum whose home is on a different node is copied not simply into the processor cache, but also into the RAC portion of memory. Locally-allocated data are copied only into the processor cache, and when evicted from the processor cache, are written back to local memory. Because the backing store for such data already exists locally, and because the RAC operates at the same speed as local memory, there is no reason to copy locally-allocated data into the RAC. The RAC size is determined statically when the machine is booted.

COMA. Another method for enhancing data locality is to use not just a portion of memory as a cache, but to treat all of memory as a cache under hardware control. In this case, the backing store migrates to where the datum is used. This type of architecture is termed a *Cache-Only Memory Architecture*, or COMA [HLH92][SJG92][HSL94]. The desired effect in this scheme is that memory will migrate to where it is needed. By not statically partitioning the amount of memory that can be devoted to caching remote data, a COMA can dynamically react to reference patterns, and can remove the burden of static data placement. For example, in a RAC, if all data accesses from a processor are to remote data, then the RAC can easily fill up. In COMA, however, the fraction of local memory that can contain remote lines is not fixed, so it can potentially accommodate such a workload more effectively.

MigRep. In contrast to the previous approaches, which perform caching at the granularity of cache lines under hardware control, we can imagine using software–namely, the operating system (OS)–to implement policies for moving data. We refer to this scheme as *Page Migration/Replication,* or MigRep. Because the OS deals only at the granularity of pages, we instead replicate and migrate data at the granularity of pages. The OS, not the hardware, is responsible for determining which pages are candidates for replication and migration. Because the OS has information about memory usage on each node, it can be selective in determining when to migrate or replicate a page, responding to dynamic information such as the amount of free memory on a node. Various techniques for OS-based migration/replication have been proposed in previous work[BFS89][CoF89][Hol89][ScD89][BSF+91][LKE91][LHE92][VDG+96].

In addition to the above approaches, another possibility is a scheme that relies on the combination of hardware and software. For example, the hardware and software may split the tasks of replication and migration. The hardware may perform some replication on its own, but surrender control to software so that the OS can migrate the data. The advantage of such a scheme is the ability to capitalize on sharing patterns that do not fit neatly into a hardware-only or software-only approach.

While a number of the above schemes are viable, and some have been implemented in real machines, there has never been a consistent and detailed evaluation of these approaches, and thus there is a lack of performance data to quantify these approaches against one another. This lack of data stems from the fact that these approaches have all be implemented on machines with different baseline assumptions, like different processors or different networks or different memory systems. Comparing machines directly against each other becomes difficult because of the number of variables that change between machines.

In order to provide some performance data, many simulation studies have been performed that compare subsets of these schemes. However, these schemes have several drawbacks:

- **The full range of hardware and software approaches have not been compared against one another.** Most previous studies have proposed a variant of one of the architectures above and compared it against some baseline machine. Some have compared COMA against a base multiprocessor with caches[SJG92], while others have compared COMA against RAC [Zhang97]. Others have compared MigRep against a base multiprocessor both with and without caches[Cox89][Bolosky89][VDG+96]. Recent studies have proposed hybrid schemes and compared them against COMA or against a base multiprocessor with caches [FaW97][HSL94][ELP+98]. Attempting to compare the results for different schemes across different studies is difficult for the same reasons that comparing machines with dissimilar underlying architectures is difficult, i.e., different hardware and cost assumptions.
- **These studies have used primarily scientific applications, rather than compute-server workloads.** Another drawback of previous studies is that they use primarily scientific applications: compute-server workloads are not considered, and the operating system and kernel effects are ignored as well. The operating system can play a key role in applications, especially if it is moving tasks around in order to perform load balancing across the different processors.
- **These studies have used high-level models of the schemes, rather than detailed, accurate implementations.** An additional concern is that previous simulation studies have all used high-level behavioral models of these different schemes, rather than detailed implementations: such models introduce the possibility of inaccuracy of results depending on the costs assumed and the contentions modeled. Moreover, with high-level models it can be difficult to determine the relative complexity of the schemes.

The goal of this research is to evaluate the full spectrum of techniques for data migration and replication (hardware-only, software-only, and hybrids) in a consistent and detailed manner on compute-server workloads. Compute-server workloads are particularly tough to deal with because the dynamic load balancing employed by the OS can cause processes to move around, and exacerbate the problems caused by lack of locality. Note that each of the schemes we have proposed above uses memory in a slightly different way to improve data locality. The RAC and COMA schemes rely on using memory as a cache under hardware control[1], while MigRep and hybrid schemes use memory as a cache either under OS control, or under control of

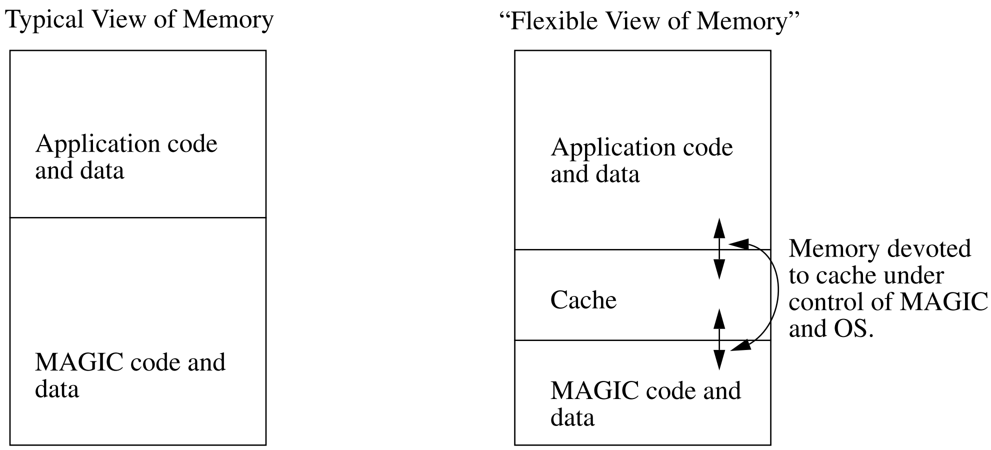


FIGURE 3. Flexible Use of Memory in the FLASH Multiprocessor. The programmable memory controller in FLASH, the MAGIC chip, allows all or part of the memory on a node to be used as cache, allowing a number of techniques for data migration/replication to be implemented. The amount of memory used for cache can be varied by the OS or by MAGIC**.**

both hardware and the OS. A fair evaluation of these schemes would require all of them to be implemented on the same base platform.

We utilize the Stanford FLASH Multiprocessor[KOH+94] for our study. FLASH is unique in that each node contains a programmable memory controller (called MAGIC, or Memory And General Interconnect Controller) and a single pool of DRAM that can be used in a variety of ways by the memory controller. By reprogramming the MAGIC chip, numerous methods for caching in memory can be supported. Figure 3 illustrates the resulting flexible view of memory MAGIC and the OS allows. Through the flexible use of memory, all of the schemes discussed earlier can be implemented and evaluated on the same base platform.

1.2 Research Results and Contributions

This thesis expands upon previous evaluation studies in a number of ways:

- While some of these approaches have been compared in previous work, this broad range of schemes (processor caching, RAC, COMA, and OS-assisted migration/replication), covering both hardware-based and software-based approaches, has never been pulled together and evaluated before.
- All of these schemes are implemented and evaluated on the same base platform. Each is a complete and working implementation, including OS modifications, rather than simulated behavioral high-level models of the protocols. This allows a more realistic, consistent, and fair comparison of these schemes than has previously been possible.

1. COMA schemes vary in the extent to which they rely on hardware, as will be discussed in the related work chapter.

- The COMA protocol evaluated in this study, COMA-F, has never been implemented before. While there have been numerous proposals and studies of various COMA architectures (requiring various levels of hardware and software intervention), and a number of variants of COMA-F proposed, this thesis presents the first complete COMA-F implementation, as well as a detailed description of many of the challenges of implementing COMA-F, many of which have never been discussed before.
- This evaluation is performed on multiprogrammed compute-server workloads, with both user and system effects included. Most previous studies have concentrated on scientific applications, and have not considered kernel references. As a result, performance under conditions with mechanisms that may stress OS locality optimizations, like process migration, has never been evaluated.
- This study proposes a novel technique, MIGRAC, for improving data locality. MIGRAC is a hybrid proposal that combines a RAC and OS-assisted page migration/replication to provide benefits of hardware and software approaches.

There are two ways in which to interpret the results presented in this thesis. First, the results can act as a guide to the desirability of implementing each of the individual schemes in this study. We present performance data comparing each of the schemes, and also discuss the complexity trade-offs of each, allowing one to decide which method would be best to implement for their particular design. A second way in which to interpret the results of this thesis is to view it as a validation of flexibility in multiprocessor design. By enabling the implementation of numerous schemes, and allowing a choice of scheme depending on the workload, FLASH-like machines can potentially provide higher performance across a range of workloads than a similar machine with a hardwired protocol engine.

Our results show that the simple RAC scheme can be implemented with low additional complexity over base cache-coherence, and is effective in improving performance (up to 64% faster than base CC-NUMA) by caching data in part of the local memory. However, the gains are quite sensitive to the size of the RAC. In particular, performance can degrade when the RAC is too small to capture the remote working set of the application, or when most of the misses are due to coherence. The COMA protocol also improves execution time (up to 14%) when the working set of the application is large and capacity misses dominate. However, it is complex to implement (both in terms of amount of protocol code required and number of instructions executed by the protocol processor), and the performance can be significantly worse if coherence misses are dominant. Both RAC and COMA are quite effective in increasing locality for both user and kernel references. However, RAC is always superior to COMA, given our base parameters and workloads. Kernel-based migration and replication requires the least changes to the base CC-NUMA protocol and does quite well (up to 33% faster than base CC-NUMA) when sharing is coarse-grain and pages are mostly read-only. We also found that the kernel-based and RAC schemes complement each other and propose a hybrid design called MIGRAC, in which kernel-based migration/replication handles coarse-grain locality decisions while the RAC protocol exploits fine-grain sharing. MIGRAC performs well (up to 57% faster than a system with just a RAC, and up to 24% faster than a system with just MigRep) and is robust across different workloads.

1.3 Organization of this Thesis

The next chapter discusses some of the issues involved in multiprocessor design, and describes the platform for this research, the Stanford FLASH multiprocessor. The features of FLASH that allow flexible use of memory for replication and migration are presented.

Chapter 3 describes in more detail the mechanisms for replication/migration under study in this work, including the challenges inherent in each scheme. It provides a qualitative comparison of the approaches, as well as a quantitative discussion of the complexities of each scheme.

Chapter 4 describes the experimental environment and workloads used in this study, and presents the performance results for each of the various replication/migration schemes. The effects of varying a number of key parameters, such as portion of memory devoted to caching, or the speed of the memory controller, are also considered.

Chapter 5 proposes and evaluates a hybrid scheme–MIGRAC–that combines the benefits of hardware- and software-controlled migration/replication.

Chapter 6 presents related work on migration and replication. These include commercial RAC designs, other COMA variants, and other OS-based/hybrid proposals.

Finally, Chapter 7 summarizes the major results from this research and discusses directions for future work.

Chapter 2
Multiprocessor Design Issues

This chapter discusses some of the issues involved in multiprocessor design. It begins by describing shared-memory multiprocessing and the cache-coherence problem in the context of bus-based multiprocessors. The CC-NUMA design and directory-based cache coherence are discussed as methods for overcoming the scalability limitations of bus-based designs. The chapter concludes with a description of the FLASH multiprocessor.

2.1 Shared-Memory Multiprocessors as Compute-Servers

Shared-memory multiprocessors have the notion of a single global address space. Processors are allowed to access any address in the global shared memory directly via loads and stores. The data is implicitly communicated: on a load or store, the hardware automatically finds and forwards the latest copy of the data to the appropriate node. This shared-memory model is desirable for a variety of reasons[Ver98]:

- Programmers are used to a single-address space since that is what is used in uniprocessor applications. In turn, familiarity with this model makes it easier to speed up large programs by parallelizing them and using the multiple processors in the system.
- Legacy uniprocessor applications for a given architecture can be run unmodified on the parallel architecture, since the number of processors and their locations are not explicitly needed by the programmer except for performance enhancement.
- The additional processors and memory can be used in different ways depending on the workload. For example, the additional memory can be used to run sequential applications whose datasets would not have fit in the memory of a single node. In addition, parallel applications can use the additional processors and memory to achieve performance speedups.

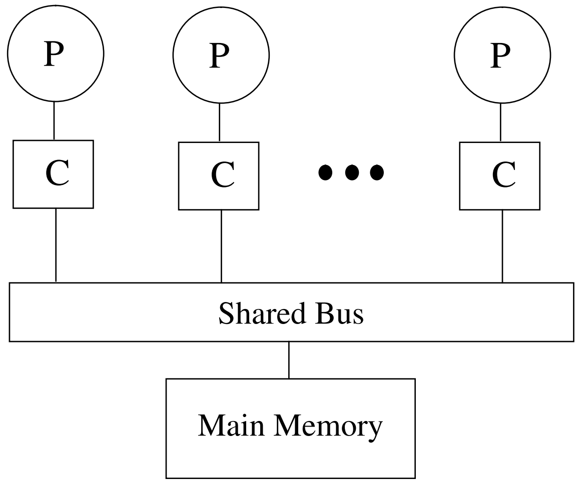


FIGURE 4. Architecture of small-scale shared-memory multiprocessors. Small-scale shared-memory multiprocessor designs feature a number of processors (P) with their caches (C) connected over a shared bus to main memory. Such machines are called UMA (Uniform Memory Access latency) because each processor can access any memory location with the same latency.

For these reasons, shared-memory multiprocessors are particularly amenable to compute-server environments. The tight coupling of resources provides high bandwidth communication between processors and memory or other resources. Different users can access different devices or memories without any knowledge of the location of such resources, and increasing the number of such resources can transparently improve performance. In order to provide such benefits, shared-memory multiprocessors require the ability to share data efficiently. We next consider how such sharing is enabled.

2.2 The Architecture of Shared-Memory Multiprocessors

Small-scale shared-memory multiprocessors use a bus to connect the processors, as shown in Figure 4. This type of architecture, in which main memory can be accessed with the same latency from any processor, is known as an UMA, or Uniform Memory Access latency, architecture. In order to prevent costly accesses to memory for each data access, caches are used to store local copies of data and promote data sharing.

The introduction of caches for replicating data leads to what is known as the cache-coherence problem. As seen in Figure 5, the cache-coherence problem is the result of allowing nodes to satisfy processor requests from a local cache, rather than satisfying all accesses directly from main memory. Without caches, all nodes go to the same main memory location for a given datum. Whenever a datum is modified, the change occurs directly in memory, and all subsequent accesses read the most recent value. In a system with caches, however, the situation is more complex, and is best illustrated by an example. Referring again to Figure 5, suppose processor 0 accesses location A and stores it in its cache (operations 1a and 1b in the figure). Now suppose processor 1 accesses location A and a copy is automatically made in its cache (operations 2a and 2b). If processor 0 now tried to write location A, it would find A in its cache, and modify it locally (operation

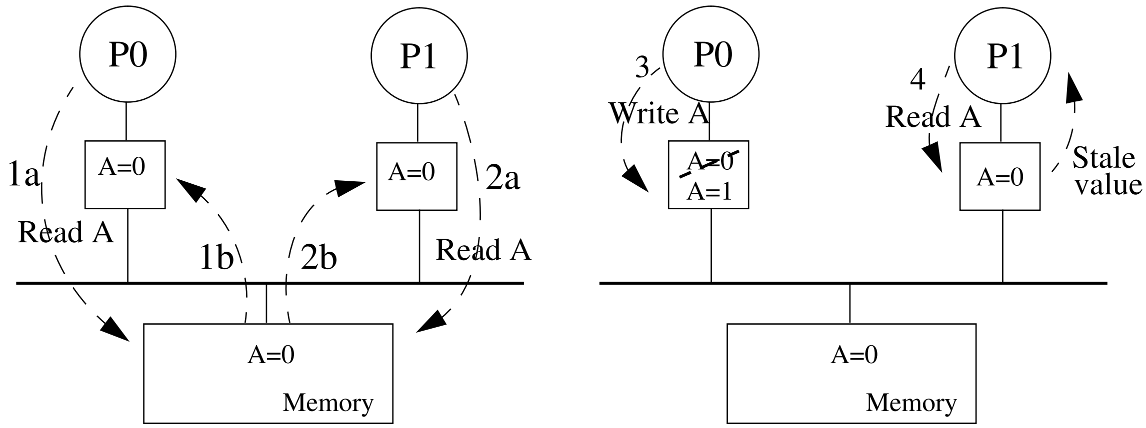


FIGURE 5. The Cache-Coherence Problem: P0 references location A. It misses in its cache and requests the latest value from memory (operations 1a and 1b). P1 also misses on location A, and also loads the latest value from memory into its cache (2a,b). Now references for A from P0 and P1 can be satisfied in their caches. If P0 writes A (3), then the next time P1 references A (4), P1 will retrieve a stale value from its cache.

3). However, that would leave processor 1 with a stale value for location A the next time processor 1 tried to read the data (operation 4). The read of A would not return the most recent value written, violating the programmer's fundamental assumption about intended behavior of his program. This behavior is probably not what the programmer expected. To prevent processor 1 from reading a stale value, communication is needed between the two processors.

A number of schemes have been proposed to deal with the cache-coherence problem in bus-based machines[McC84][PaP84][SwS86][TSS88]. The general method involves tagging cache lines with state indicating whether or not a processor owns a given cache line, and using the bus to broadcast all transactions to all processors. Each processor snoops on the bus probing for transactions related to addresses in its cache. When an address match occurs on the bus, the processor performs the appropriate action on the value in its cache. In a multiple-reader, single-writer, invalidation-based design, multiple concurrent readers for a line are allowed, but only one writer is allowed. In the MESI[PaP84] protocol, each line can be in one of four states on each node: Modified, indicating that the line is dirty on that node; Exclusive, indicating that the line is cached clean (with respect to memory) on that node, but is the only cached copy in the system; Shared, indicating the line is clean on this node, and is also in other node's caches; and Invalid, indicating that this line is currently not resident on this node. Each read miss in the processor cache (i.e. line is Invalid) goes onto the bus broadcast to request the datum, and someone (either a cache, if the line is dirty, or main memory, if the line is clean) supplies the data. For a write request to succeed, a node must not only have that location in the cache, but also be the sole owner of that line (i.e. it must be Modified or Exclusive). If a node N is not the sole owner, then the request goes on the bus, and other copies are invalidated, making node N the sole owner. If node N did not have a copy of the data initially (i.e. the state was Invalid), then the request

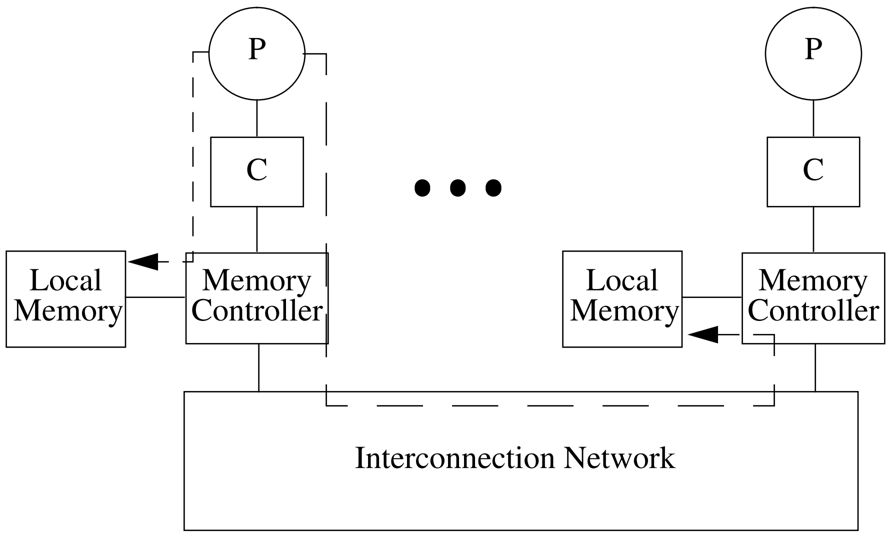


FIGURE 6. Architecture of a CC-NUMA Multiprocessor. Each node contains a processor (P), its associated cache (C), and a portion of the globally-distributed main memory. Nodes are connected by a scalable interconnection network. The dashed lines show the paths of local and remote memory references. Local references incur less latency than remote references. The memory controllers cooperate to provide coherent caches and coordinate accesses to memory.

would go on the bus, and either the current owner or memory would surrender the data, while invalidating all other cached copies. Subsequent readers of that line will miss in their processor caches and their requests will go onto the bus in order to get the most up-to-date copy. Because all processors snoop the bus on all accesses, it is easy to find up-to-date copies of data or perform invalidations.

While the above snooping scheme works very well for bus-based multiprocessors, buses do not scale well to large numbers of processors. At large numbers of processors, the bus is saturated with requests and becomes a bottleneck. Moreover, it becomes difficult to regulate the electrical characteristics of the bus properly at high speeds, and signal noise is a problem.

As an alternative to buses, scalable cache-coherent machines typically use a Cache-coherent Non-uniform Memory Access latency architecture (CC-NUMA). As depicted in Figure 6, a CC-NUMA machine is divided into nodes which are connected by a scalable, high-performance interconnect. Each node consists of one or more processors plus associated caches and a portion of the globally-distributed main memory. Within a node, multiple processors may be connected by a bus, but between nodes there is a high-performance network (for example, a two-dimensional mesh interconnect). Because memory is distributed throughout the machine, rather than centralized, the access latency is non-uniform, and varies depending on where a datum is physically located. Moreover, the use of a network rather than a bus to connect the processors removes the bus as a serialization point, and there is potential for scalability to large numbers of processors. A number of academic prototypes such as Alewife[ABC+95], DASH[LLJ+92], and FLASH[KOH+94], and commercial machines like the HP Exemplar [BrA97], Sequent STiNG[LoC96], Data General NUMALiiNE[Dat97] and SGI Origin 2000[LaL97] have been built based on this topology.

The use of the CC-NUMA design leads to a number of issues that do not arise in bus-based designs. In order to understand the complexity of the issues involved in implementing schemes for data migration and replication, it is important to first understand the basics of cache-coherence in CC-NUMA machines, and how it differs from the SMP case.

2.3 Directory-Based Cache Coherence

CC-NUMA multiprocessors do not use a bus to communicate between processors, and generally do not provide a mechanism for broadcasting transactions to all processors simultaneously. Instead, communication occurs via messages that are sent from a requestor node to a receiver node. This lack of broadcast makes cache coherence more difficult to maintain for two main reasons:

- Processors do not see actions at the same time. In UMA, once a processor locks the bus, all processors observe its operations atomically. In CC-NUMA, messages must be sent individually to all processors involved in a given transaction, and the entire transaction is not atomic. Thus, provisions must be made for multiple outstanding requests for the same line from different processors.
- Data transfer is not a simple matter of taking control of a bus and having the appropriate memory or cache return the data. Instead, a request message must be sent directly to the node containing a datum. That node must then return the data via another message.

These concerns force a CC-NUMA cache-coherence protocol to maintain some data structure that somehow enforces atomicity and tracks the sharers of a given datum. One common solution is to use a directory to store this information: a protocol utilizing this solution is termed a *directory-based cache-coherence protocol*.

In a directory-based cache-coherence protocol, a directory entry is maintained for every line in memory. This directory entry contains a list of all sharers for that line. The directory is stored at the *home* of that line, which is the node from whose memory the line is allocated. Whenever a read request occurs for a line, the home is consulted. If the data is clean, the line is read from the home memory and returned to the requestor. If the data is dirty in some processor's cache, the home forwards the request to the owner, who satisfies the request directly and then changes its status to shared. The requestor is then added to the sharing list. If a write request occurs for a line, the home is again consulted. If the line is clean, all sharers of that line are notified that the line is to be written. These sharers invalidate their copies of the line, and the home reads the data out of memory and forwards it to the requestor. The directory then designates the data as dirty and sets the requestor as the sole sharer. There is often some state in the directory that indicates when a transaction is in progress–if another request occurs while one is already in progress, this state prevents the second access from occurring, essentially preserving atomicity of transactions.

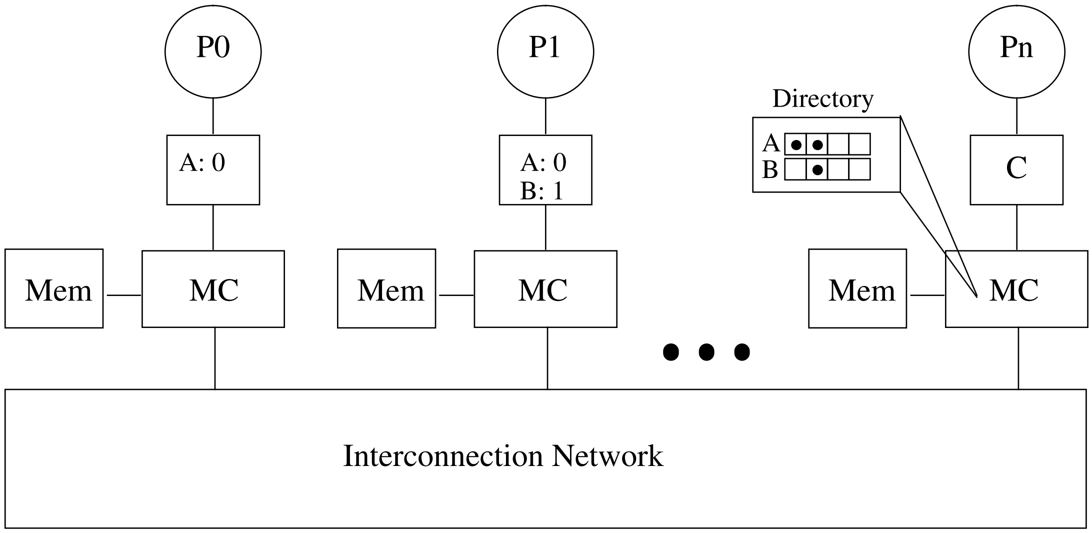


FIGURE 7. Directory-Based Cache Coherence using a Full-Map Directory Scheme. In a Full-Map scheme, a bitvector encodes the identity of the sharers. In this case, Pn is the home for locations A and B. Location A is cached in P0's cache and in P1's cache, as indicated by their bits being set in the directory at Pn. Location B is cached only by P1, so only P1's bit is set in the directory. In more complicated directory schemes, a data structure different from a bitvector lists the identities of the sharers.

The first directory-based protocol proposed for shared-memory machines was a full-map protocol [CeF78], depicted in Figure 7. In this scheme, the sharing list is a bitvector, with one bit for each processor in the system. When a processor shares a line, the appropriate bit is set. When a line is invalidated from a processor, that processor's bit is reset. While conceptually simple, a full-map scheme does not scale well to large numbers of processors: the storage requirements for such a directory increase proportionally to the square of the number of processors [Web93]. As a result, a number of other protocols have been developed instead [Sim92][SCI93][Web93][Hei98] to try to achieve better scalability as the number of processors increases.

2.4 Locality Issues in CC-NUMA Architectures

The complexity of implementing a directory-based cache-coherence protocol is one distinction between CC-NUMA and UMA. Another important difference involves the non-uniformity of memory latencies. In UMA, all memory is equally far from each processor, so that all memory accesses are the same latency. In CC-NUMA, however, the latency to access data on a remote node is larger than that to access data on a local node, because the remote data is farther away. The ratio of the remote latency to local latency can be as low as 2:1, as in the SGI Origin 2000 [Lal97], or as high as 10:1, as in the Sequent STiNG [LoC96]. Because poor memory access time increases the memory stall time component of execution time, data locality becomes crucial. In CC-NUMA, unlike UMA, there is a choice of where to place the data, and the wrong choice can negatively impact performance. For high performance, it is valuable to have data as close as possible to the requesting node.

In order to provide good data locality, caching is usually combined with schemes that perform migration and replication in memory. A number of current commercial architectures implement some form of cache coherence and augment it with some replication in memory [LoC96][BrA97][Lal97]. The forms of replication and cache coherence employed are usually decided at design time so that support for them can be implemented efficiently in hardware. Because of the varying nature of these architectures it is difficult to compare their performance–each has a different memory system, a different node design, and different network implementations. A machine design that allows multiple protocols for cache coherence and replication is thus desirable for fairly comparing different approaches. Moreover, different mechanisms for locality enhancement are effective under different circumstances, and a flexible design allows use of different schemes for different workloads. The Stanford FLASH multiprocessor is one such design, incorporating a flexible memory and directory controller to allow multiple cache-coherence and replication strategies to be implemented on the same base platform.

In order to provide a meaningful comparison between different approaches, a machine must not only support multiple replication/migration strategies, but must also provide performance that is competitive with comparable hardwired machines. To understand how FLASH is capable of supporting multiple replication/migration strategies efficiently, it is important to understand the architecture of FLASH, so we discuss the FLASH design in detail in the next section.

2.5 The FLASH Multiprocessor

In this section we discuss the FLASH multiprocessor. First, we discuss the overall architecture of FLASH. Next, we describe the features of FLASH that allow it to flexibly implement various schemes for data replication and migration.

2.5.1 Architectural Description

The Stanford FLASH (FLexible Architecture for SHared memory) Multiprocessor [KOH+94] is a directory-based cache-coherent shared-memory multiprocessor built around the MIPS R10000 processor and the MAGIC (Memory And Global Interconnect Controller) custom node controller chip. Each node contains an R10000 processor with its associated cache, a portion of globally shared memory, I/O devices, and the MAGIC chip. A diagram of the FLASH architecture is shown in Figure 8.

The MAGIC chip is central to the FLASH architecture. MAGIC behaves as a network interface, I/O interface, and memory controller, communicating with the I/O subsystem, the network, and the DRAM on behalf of the R10000 processor. When the processor needs to interact with these components, it issues a load or store instruction to MAGIC. MAGIC interprets the load or store appropriately and coordinates with the appropriate subsystem, returning the data to the processor. A logical block diagram of the MAGIC chip is

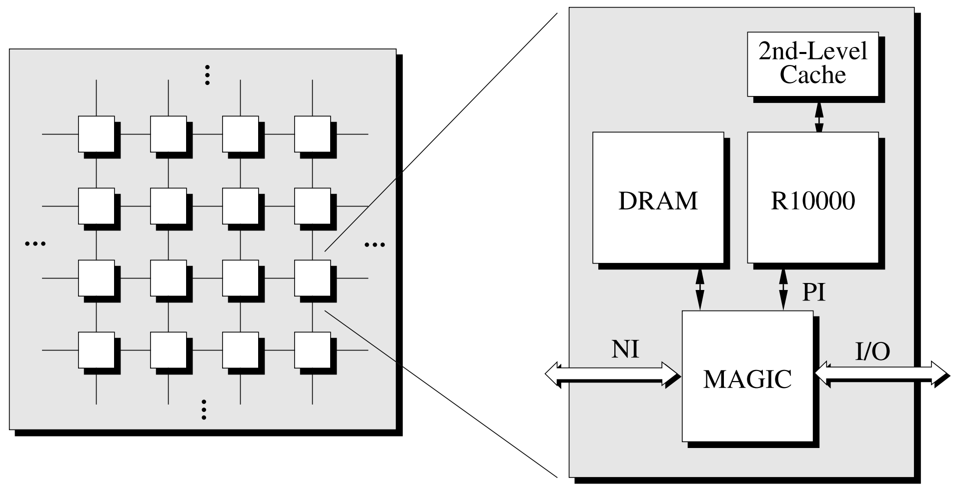


FIGURE 8. The FLASH Architecture. The MAGIC chip integrates the CPU, memory, I/O, and network interfaces**.**

shown in Figure 9. Essentially, MAGIC is responsible for performing accesses to memory on behalf of the processor as well as retrieving data from other nodes.

Inputs to MAGIC can come from three sources: the processor, the network, and the I/O subsystem. The processor interface (PI) maintains the queue of requests from the processor, the network interface (NI) maintains the queue of requests from the network, and the I/O interface (IO) maintains the queue of requests from the I/O subsystem. Prior to scheduling a request from a given queue, the resource requirements for that queue must be determined. For example, because a request from the PI may generate an outgoing network request, there must be space on the NI outgoing request queue in order to schedule any PI request. The detailed queue requirements are beyond the scope of this thesis and are described in [Hei98][Kus97].

The flow of a processor request in FLASH proceeds as follows. The CPU emits a request. If it is a cache miss, it is forwarded to the MAGIC chip and stored in the PI incoming queue. When MAGIC is able to schedule that request, it pulls the request off the PI incoming queue. If the request is to a datum in local memory, MAGIC initiates a memory request for that datum. Eventually, the memory system returns the datum to MAGIC, and MAGIC returns the datum to the processor. If the request were located in a remote memory or remote cache, MAGIC forwards the request over the network to the appropriate node. The address of the datum tells whether it is locally- or remotely-allocated, but directory accesses may be required to determine whether or not it is cached remotely.

The flow of a network request is very similar to that of a processor request. A network request is received at the NI incoming queue. When the request is ready to be scheduled, MAGIC pulls it off the NI queue. If a directory access is needed to determine the location of the datum, then one is performed. If the datum is in memory, MAGIC submits a memory request. If the datum is in the processor cache, MAGIC initiates an *intervention* request to the processor. This intervention requests the specified datum from the processor cache. If the datum is not on this node, the request is forwarded to the node that contains the datum. After MAGIC receives the datum from memory or the processor cache, it sends a reply back to the requestor. The requestor's MAGIC chip receives the reply, forwards the datum to the processor, and performs any directory processing that may be required.

IO requests are handled in the same manner as PI requests and NI requests. IO requests are emitted from the I/O subsystem and are placed in the IO queue until MAGIC can service the request. MAGIC determines the location of the requested datum through a directory lookup. If the datum is in local memory, MAGIC submits a request to memory. If the datum is in the processor cache, MAGIC issues an intervention to the processor. If the datum is on a remote node, the request is forwarded to that remote node. In any case, when MAGIC eventually receives the datum, it returns the datum to the I/O subsystem.

As the above descriptions demonstrate, MAGIC plays a key role in the architecture of FLASH, performing a range of tasks from coordinating communication between nodes to acting as the memory controller. One of the most important duties of the MAGIC chip is to implement the cache-coherence protocol. MAGIC is unique from controller chips in other distributed shared-memory (DSM) machines in that the cache-coherence protocol is not hardwired. Instead, MAGIC contains a RISC core (the protocol processor, or PP) which allows the protocol designer to run different protocols simply by downloading a new protocol onto MAGIC. This capability is invaluable for protocol research, and is also useful for debugging purposes. The PP is logically contained with the control pipeline portion of Figure 9. A number of protocols have been proposed and implemented for FLASH, including a full-map directory[HS+99], the Dynamic Pointer Allocation Scheme[Sim92][HS+99], and the Scalable Coherent Interface [SCI93][HS+99]. Other invalidation-based schemes like LimitLESS[CKA91] are also possible.

A cache-coherence protocol written for the PP consists of a number of procedures called *handlers*. A handler is a sequence of instructions that operates in response to an incoming message from the processor or network/IO. All handlers run until completion, although if certain resource limitations occur, a handler may have code that writes state out to memory and deschedules itself temporarily: in all cases, only one handler runs at a time on the PP.

Clearly, a protocol written for the PP may be slower than a hardwired protocol, which would impact the ability of FLASH to fairly compare different protocols. However, MAGIC contains a number of optimizations that attempt to mitigate some of the potential overheads, and allow FLASH to perform well relative to other DSM machines with different protocols[Hei98]. First, while protocol instructions and data are stored in main memory just like application data and instructions, FLASH contains MAGIC instruction and data caches in order to alleviate memory bandwidth needs[1]. Second, MAGIC contains specialized instructions for common protocol operations, like bitfield manipulations or cache-line-sized reads from memory. In addi-

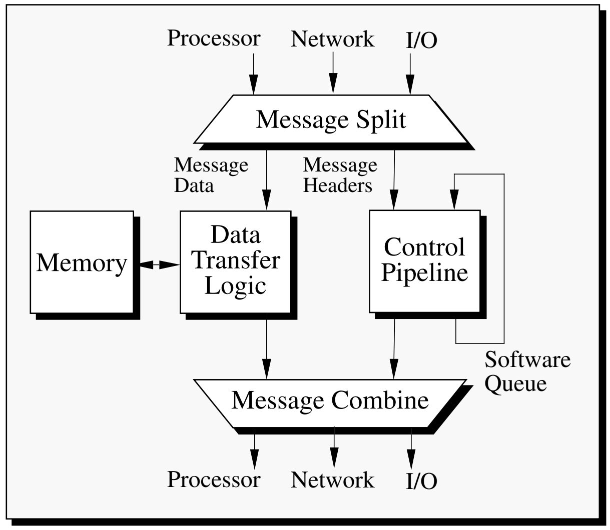


FIGURE 9. Block Diagram of the MAGIC Chip.

tion, MAGIC separates control and data transfer to allow pipelining of requests, allowing MAGIC to effectively work on multiple requests in parallel, improving MAGIC request bandwidth.

Separation of control and data transfer is achieved through a combination of dedicated hardware and special instructions. In order for MAGIC to access memory lines, the PP contains special non-blocking instructions that load from or store to memory in cache-line-sized chunks. Thus, the PP need only issue one load or store instruction per memory line and then continue processing the handler, rather than cycling through a line issuing blocking loads and stores on a word-by-word basis. Special cache-line-sized hardware structures called *data buffers* store the data as it is being received from or written to memory, removing the burden of providing data storage from the PP.

The separation of control and data transfer is combined with pipelining to increase the MAGIC request bandwidth. Control pipelining is performed by separating protocol processing into Inbox, PP, and Outbox stages [Kus97], as shown in Figure 10. Each unit is specially tuned so that its portion of protocol processing proceeds quickly and efficiently. To illustrate how this division of labor provides efficient pipelining of MAGIC requests, we describe each of these units in turn.

Inbox. When a request enters MAGIC, it is pre-processed in the Inbox. The Inbox has several functions. The Inbox arbitrates among the various queues serving MAGIC, determining which incoming requests have their resource requirements satisfied and are ready for processing. If a request may require a memory operation, the Inbox issues the memory operation to the memory controller. Issuing a memory reference before the handler for the address is invoked allows overlapping of the memory access latency with the handler

1. The instruction cache is on-chip, but the data cache is off-chip.

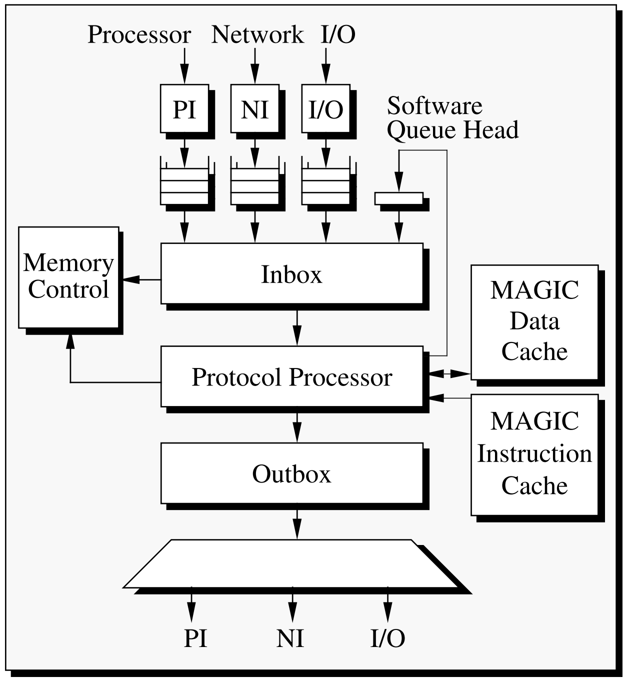


FIGURE 10. Block Diagram of the Control Macropipeline. The Inbox arbitrates among the queues servicing MAGIC to choose a request to process. The Protocol Processor (PP) executes the instructions of the handler. The Outbox processes sends on behalf of the PP. The MAGIC data cache is off-chip**.**

latency, thus hiding some of the latency of the handler. Because a memory operation occurs before any state for that line is checked, it may be unnecessary. A memory operation initiated by the Inbox is thus termed a *speculative memory operation*. The other main function of the Inbox is to parse the message type and choose the PC of the appropriate handler. A special hardware lookup table called the JumpTable, indexed by the message type, provides the PC of the handler that is to run on the PP.

Protocol Processor. After pre-processing by the Inbox, the request is passed into the PP. The PP executes the handler code beginning at the PC indicated by the JumpTable, while any speculative memory operation issued by the Inbox occurs in the background. When the memory operation is complete, its data buffer is automatically filled without interrupting the PP. A handler runs in the PP until all actions are completed—the handler cannot be pre-empted by requests from other nodes or from the processor. As a result, a handler has exclusive access to protocol data structures stored on that node, and external events cannot modify those data structures until their corresponding handlers run, alleviating the need for explicitly synchronized access to protocol data.

The Outbox. The final stage of processing occurs in the Outbox. When the PP needs to send data to the network, IO, or the processor, it constructs the outgoing MAGIC message header in its general-purpose registers. The contents of these registers, as well as message send type bits, are sent to the Outbox. The Outbox determines the proper destination unit for the outgoing message, reformats the message header accordingly, and passes the reformatted header to the destination interface unit for further processing. By accepting the send on behalf of the PP, the Outbox allows the PP to issue a send and continue processing, even if the destination interface unit is busy.

The control pipelining allows the MAGIC chip to process up to three requests at once. While the PP is processing the current handler, the Outbox can be completing the send of the previous handler, and the Inbox can be choosing the PC of the next handler. The data buffers provide management of the cache lines so that the Inbox, PP, and Outbox can concentrate on the control aspects of a message transaction.

Another novel feature of FLASH compared to other flexible architectures [RLW94] is that the connections to memory, network, and IO from MAGIC are point-to-point, rather than bus-based, allowing faster interactions than a bus-based design would allow. The drawback is that all requests from the processor or from the network/IO are serialized through MAGIC, so that MAGIC can potentially become a bottleneck in high-contention situations. Previous work has shown that MAGIC's pipelining and ISA changes are effective at alleviating bottlenecks and making performance competitive with a hardwired implementation [HKO+94][Hei98].

2.5.2 Deadlock avoidance in FLASH

One additional implementation detail that impacts protocol complexity is deadlock avoidance. *Deadlock* occurs when operations are still outstanding but all system activity has ceased [CSG99]. Avoiding deadlock requires a contract between the protocol writer and the hardware: the hardware must provide mechanisms that are general enough to prevent any deadlocks from occurring, and the protocol must be specified in such a way that any deadlock can be resolved using the provided hardware mechanisms.

Deadlock typically occurs because of resource limitations[CSG99]. For example, if an incoming request requires usage of some resource, but that resource is not available, then there must still be some mechanism for responding to the request, to allow the requestor to make forward progress. There are numerous resources allocated for each incoming request, e.g., data buffers or network queue space. Designing a protocol that is correct in the absence of such constraints is difficult enough because of corner cases that can occur with non-atomic protocol transactions. Adding resource limitations makes correct protocol design even tougher. Because our protocols are intended to run on FLASH, and are not high-level models of protocols, they must deal with these issues using the provided hardware mechanisms. We defer discussion of specific deadlock examples until the protocols we evaluate are fully described, and instead just give a brief overview of deadlock avoidance in FLASH. For a more detailed discussion of the some of the issues concerning deadlock in FLASH and in shared-memory multiprocessors in general, see [Kus97][Hei98].

FLASH maintains separate request/reply networks (specifically, separate virtual lanes) in order to avoid deadlock resulting from request/reply cycles in the protocol, sometimes referred to as *fetch deadlock*

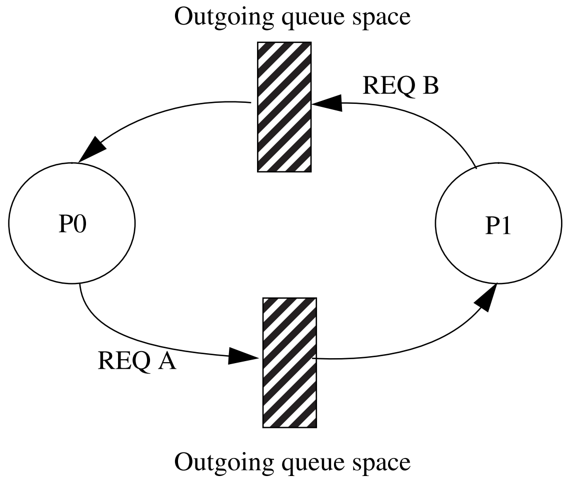


FIGURE 11. Fetch deadlock. Node P0 submits a request (REQ A) to node P1, and P0's outgoing network queue is now full. Before REQ A reaches P1, P1 submits a request (REQ B) to node P0, filling node P1's outgoing network queue. P1 now cannot send a reply to P0 because its outgoing network queue is full, and similarly, P0 now cannot send a reply to P1. Neither node can make forward progress, resulting in deadlock**.**

[CSG99]. Figure 11 demonstrates a simple example of fetch deadlock. In Figure 11, P0 sends a request to P1. This request consumes outgoing queue space on P0. Suppose this queue space is now full, so that P0 can send no more messages. Now suppose that P1 has sent a request to P0, and that now P1's outgoing request queue is full. P1 would like to generate a data response to P0, but cannot because it has no outgoing queue space. P0 would similarly like to generate a data response to P1, but also cannot because it has no outgoing queue space. Neither node can satisfy the other node's request, so neither node can make forward progress, resulting in deadlock. The trouble arises because responses consume the same outgoing queue resources that requests consume. If we instead provide two separate lanes, one for requests and one for responses, then responses will never get stuck behind requests, and we eliminate this deadlock possibility. This solution relies on a protocol in which requests can cause responses to be generated, but responses generate no other message traffic. Because responses can always be sunk, we are guaranteed that eventually each request will be satisfied. While separate lanes alleviate request/reply deadlock possibilities, most cache-coherence protocols do not provide strict request/reply semantics for performance reasons[CSG99], and other deadlock scenarios can occur. Two common deadlock conditions involve reply/reply and reply/request cycles.

A *reply/reply* cycle occurs when a protocol reply transaction generates another reply. Since most deadlock-free implementations require replies to be sunk to guarantee forward progress, a reply that requires generation of another reply can cause deadlock. For example, suppose all nodes are unable to generate outgoing replies and thus none can sink incoming replies to make forward progress: this is a deadlock situation. Similarly, a reply that can generate another request (a *reply/request* cycle) can also cause deadlock, if all nodes are unable to sink replies because they cannot generate the needed requests. These situations are approached by careful resource management within the protocol. An integral component of the deadlock avoidance strategy is an additional incoming queue to the PP called the Software Queue (SWQ), shown in Figure 9. The SWQ is essentially a list of requests that have been suspended for some reason (typically because of a lack of resources) and need to be continued when their resource requirements have been satisfied. For example, suppose an incoming NI reply may need to generate an outgoing request. In order to avoid deadlock, MAGIC is not allowed to wait until outgoing request queue space is available before scheduling the NI reply. Instead MAGIC schedules the reply. While the PP is processing the reply, if it determines that outgoing request queue space is needed, then it checks if that space is available. If that space is not available, the reply is placed on the SWQ and descheduled. Periodically, the SWQ is rescheduled. The suspended reply handler runs, and if the queue space requirements are met, then it completes. Otherwise, it is again placed on the SWQ.

Another important feature of the deadlock avoidance strategy is negative acknowledgments (NACKs). If a request that has been scheduled cannot be completed due to resource limitations, it can be NACKed. For example, in many cache-coherence protocols an incoming request may need be forwarded to a remote node. This is a scenario in which a request can cause another request. It is not a valid solution to schedule an incoming request only if outgoing request queue space is available: if every node in the system were somehow unable to generate outgoing requests, then none could schedule incoming requests, and none would make forward progress, resulting in deadlock. The solution is to schedule incoming requests as long as there is reply space available and make the NACK a reply message. Since replies are guaranteed to be sunk in order to avoid fetch deadlock, scheduling incoming requests only if outgoing reply space is available does not cause deadlock. If the PP, while processing the incoming request, discovers that it does require outgoing request space, it checks for outgoing request queue space. If no such space exists, the request is NACKed and then retried by the requestor. Because the NACK is a reply, and we have guaranteed that there is outgoing reply space, we are guaranteed to be able to send the NACK.

Note that NACKs can only be used with requests (as opposed to replies), for two reasons. First, a NACK is itself a reply: to avoid a reply/reply cycle and possible deadlock, we cannot guarantee that we will be able to generate a NACK in response to an incoming reply. Second, a requestor typically keeps enough state so that it is able to regenerate a request if the request does not complete. For example, there is a table at the requestor that stores all outstanding references by that node, so if a request is NACKed, the requestor can easily regenerate the request. For replies, typically no state is kept at the node generating the reply. As a result, if the reply were NACKed, the replying node would be unable to regenerate the request anyway.

2.5.3 FLASH System Software

One final detail relevant to our discussion of FLASH involves the system software. The FLASH machine runs a modified version of the SGI IRIX operating system. By running a commercial operating system, FLASH is able to leverage off existing applications. The use of a commercial OS also allows FLASH to run both dedicated scientific applications as well as compute-server workloads. Details of the FLASH system software strategy, particularly those related to fault containment issues, can be found in [KOH+94][CRD+95][BDG+97].

2.6 Summary

There are many issues involved in implementing a cache-coherent shared-memory multiprocessor. Rather than hardwiring a particular solution to the cache-coherence problem, the FLASH machine provides a test-bed for implementing and evaluating various cache-coherence protocols. The flexibility of FLASH can also be used to implement various schemes for caching in memory, and for this reason we have chosen it to for our study. Now that we have described the base architecture for our evaluation, we can discuss the specific schemes that we have chosen to study and their implementations on the FLASH machine. This discussion is the topic of the next chapter.

26

Chapter 3
Techniques for Improving Data Locality

The goal of this chapter is to describe in more detail our proposed techniques for improving locality. One of the important contributions of this work is a detailed implementation of a number of different data migration and replication schemes, so we concretely discuss our techniques in detail. We begin by introducing terminology that will be used throughout our descriptions. We then describe each of our protocols: base cache-coherence, base cache-coherence plus RAC, COMA, and page migration/replication. We define the design space of these protocols and characterize them accordingly. Finally, we discuss the complexity issues involved in each of the protocols and give a quantitative description of the overheads seen.

3.1 Terminology

In order to simplify the discussion of the various protocols, we now define some terms that will be used throughout our descriptions.

3.1.1 General Terms

Home node. In a directory-based cache-coherence protocol, one node is responsible for storing the directory information for a given cache line. This node is termed the *home* node, and is consulted by remote nodes seeking that cache line. The identity of the home is encoded in the physical address of the cache line, and is typically the node from whose memory the page containing the cache line is allocated. Because the home serves as the backing store for a given cache line, when a line is dirty at a remote node's processor cache, and is subsequently evicted, it is written back to the home node. The home is thus a static placeholder for clean data and for directory information. Note that the definition of home node changes in a COMA design– the differences will be discussed when we describe our COMA protocol.

Owner node. In home-based cache-coherence protocols, the home memory keeps a copy of a clean data line. If that line is dirty, it can reside either in the home cache or in a remote cache, and the home memory no longer has an up-to-date copy of the line. The node containing the most recent copy of the line is called the *owner*.

Local miss. A cache miss to a datum for which the requestor is also the home is called a *local miss*, because the line is locally-allocated. *Local data* refers to data for which the requestor is home.

Remote miss. A cache miss to a datum for which the requestor is not the home is called a *remote miss*. *Remote data* refers to data for which the requestor is not home.

Migration/Replication. *Migration* refers to the process of moving the backing store for a datum. For example, suppose the home for a line were P0, and for locality purposes the application would benefit if the home were at P1. Migrating the line means moving the backing store (i.e., the home in a standard CC-NUMA design) to P1, thereby moving the data into memory there. *Replication* simply refers to copying a datum where it is needed. The backing store for the line remains at the home. For example, caching replicates a line from its home node into the requestor's cache. These definitions are slightly different for COMA, and we will discuss this issue later.

HeadLink entry. As mentioned in Chapter 2, in most directory-based cache-coherence protocols, for every line in memory, the home of the line keeps the directory information, which typically includes the sharing list and other state bits. In FLASH, part of the directory information is stored in a structure known as the *headLink entry*. The headLink entry contains state bits for the cache line, and also contains a pointer to any additional state that may be required by the protocol, like the sharing list[Hei98]. The headLink entries for all of the lines in memory are stored in an array indexed by the address of the line. Each headLink entry is 64 bits, although most protocols do not use all 64 bits. If more directory information is needed, more memory must be allocated for that purpose.

Latency/Occupancy. Two terms related to utilization of the PP within MAGIC need to be defined to allow us to discuss the complexity of various protocol operations. Latency is defined as the time between when the PP begins executing a handler and when it generates a reply to this request. In contrast, occupancy is the time between when a PP begins executing a handler and when it finishes executing that handler. There might be bookkeeping after the generation of the response, contributing to occupancy but not latency.

3.1.2 FLASH Request/Reply Terminology

In order to understand the various message types and handler names that are used in FLASH we must first describe the message naming conventions.

Get/Put, GetX/PutX, Upgrade/UpgradeAck. The naming convention used in FLASH is that the request to MAGIC generated on a processor read miss is termed a *Get* request. The response to a Get is a *Put*. When a node wishes to write a datum, it must first obtain exclusive access to that line by submitting a *GetX* request (Get eXclusive). The response to a GetX is a *PutX*, containing the requested datum. If the node wishes to write a datum, and already has a clean copy of the datum, then it still needs exclusive access, but does not need to be sent a copy of the datum. The message indicating that a clean, shared copy is to be upgraded into an exclusive copy is termed an *Upgrade* request. An Upgrade is answered by either an *UpgradeAck* or a PutX: the UpgradeAck response does not contain data, while the PutX does.

Replacement request. When a reply datum comes to a node, and the space it is supposed to occupy in the cache already contains another dirty datum, a cache conflict is said to have occurred. This dirty datum must be evicted from the cache and sent to another node (typically the home/backing store) to be sunk. The message containing the evicted data is termed a *replacement request*, or a *writeback.*

Replacement hint. When a cache conflict occurs and the line to be evicted from the cache is clean, then the home is guaranteed to contain a clean copy of the datum (except for COMA). Thus, the node evicting the datum need only send a notification request to the home, informing the home that the evicting node no longer contains that datum. The notification message, which does not contain data, is called a *replacement hint*.

3.1.3 FLASH Handler terms

In order to aid understanding of the protocol implementations, we will briefly describe FLASH handler terminology. Recall that in FLASH, a handler is a sequence of instructions that runs on the PP within MAGIC in response to a request or reply received by MAGIC. This code is essentially microcode. We sometimes refer to handler code as *firmware*, as opposed to hardware or software, because it is software that is at a lower level of abstraction from system software and application software.

Protocol handlers typically have a three-part name. The first part conveys the source of the request (PI, NI, IO, or SW). The second part indicates whether or not the node running the handler is the home of the requested datum (i.e. local vs. remote), and the last part indicates the function of the handler (e.g. Get for a read request and GetX for an exclusive ownership request).

For example, when a processor misses on a read request to a datum for which it is the home, the handler that would run on the PP is PILocalGet: PI to reflect the processor as the source of the request, Local because the node running the handler is the home of the data, and Get because it is a read request. If this request is unable to be satisfied at the home, then it is forwarded to a remote node, and the handler that runs at that node would be NIRemoteGet: NI because the request originates from another node, Remote because the node running the handler is not the home of the data, and Get because it is a read request. The reply sent to the requestor would be NILocalPut: NI because the reply originates from another node, Local because the node running the handler is the home of the datum, and Put because the message is a reply is to a Get.

With this naming convention in mind, it will be easier to discuss the implementations of our protocols. First, we explore the design space for replication/migration protocols in FLASH, then we will describe them in more detail.

3.2 Design space for replication/migration

Each scheme for migration and replication can be characterized according to how it answers several important questions:

1. Where does replication/migration occur?
2. Who decides when to replicate/migrate?
3. What happens when a conflict occurs?
4. What is the resulting view of memory?
5. What are the potential benefits over straight cache coherence?
6. What are the challenges in building this scheme over just using cache coherence?

Let us consider each of these questions in more detail.

Where does replication/migration occur? Data can be moved or copied into just the processor caches, into memory on the requesting node, or into both memory and cache on the requesting node.

Who decides when to replicate or migrate? Who enforces the migration/replication policy? Is the underlying cache-coherence protocol responsible for choosing when to replicate and when to migrate, is the operating system responsible, or is it a combination of both? Moreover, is replication/migration *eager* or *delayed*? In other words, does a datum (whether it be a cache line or a page) get copied or migrated as soon as it is requested (eager), or after repeated references to that datum (delayed)? Who makes the policy decision: the OS or the protocol? What flexibility is there in choice of policy?

What happens when a conflict occurs? On a data reply, is the location that the reply data will inhabit (whether memory or cache) already occupied by valid data? On such a conflict, the data currently in the cache/memory will need to be written back to the home (or a replacement hint must be sent for that line, if it is clean). Is the writeback sent to the home guaranteed to be sunk at the home?

What is the resulting view of memory? Memory can be thought of as backing store for application code/data, and backing store for protocol code and data. Is memory also used as a cache? If so, is the memory statically or dynamically partitioned for this purpose?

What are the potential benefits over base cache coherence? What types of locality optimizations are used in order to improve performance over simple cache coherence?

What are the challenges in building this scheme over just using cache coherence? There is a complexity trade-off in performing replication/migration in memory, either at the protocol level or at the OS level. What are some of the sources of this complexity in each of the schemes that we discuss?

We will now discuss in detail each of our approaches: base CC-NUMA using Dynamic Pointer Allocation, CC-NUMA + RAC, COMA, and MigRep. We begin by giving a brief overview of the technique, followed by some sample protocol transactions. We then characterize them in our design space, and describe the complexity issues involved with the respective approach.

3.3 Base Cache Coherence: Dynamic Pointer Allocation

Our baseline approach for replication/migration is simply to use coherent caches. A number of commercial machines and academic prototypes use coherent-caches in a CC-NUMA design to improve locality, including DASH[LLJ+92], Alewife[ABC+95], SGI Origin 2000[LaL97], Hal[WGH+97], HP Exemplar[BrA97], Data General NUMALiiNE[Dat97] and Sequent NUMA-Q[LoC96]. The cache-coherence protocol chosen as the base for this study is the Dynamic Pointer Allocation Scheme (DynPtr). DynPtr was proposed in [Sim92] to scale to larger numbers of processors than is possible with a full-map directory scheme. The implementation of DynPtr on FLASH was written by Mark Heinrich [Hei98].

DynPtr is a multiple-reader, single-writer, invalidation-based protocol, similar to the DASH protocol [LLJ+92]. Similar to a full-map scheme, the home keeps track of all sharers of a line through a directory data structure. DynPtr achieves its scalability by keeping track of the sharers using a singly-linked list maintained at the home, rather than through a bitvector. While the use of a singly-linked list to store the sharing list adds to the complexity of protocol transactions, particularly those for adding and removing sharers from the list, at large machine sizes the enhanced scalability can outweigh the costs [HS+99][Hei98]. The directory information is split in DynPtr: frequently-accessed state, such as the first sharer in the sharing list, whether the line is clean or dirty, or whether there are pending transactions on that line, is kept in the headLink, while the remainder of the sharing list is kept separately. A conceptual picture of the headLink data structure is shown in Figure 12, and a brief description of each of the fields is shown in Table 1. It is not critical to understand the purpose of each of the fields. Rather, the headLink is shown simply to illustrate the separation of frequently accessed state from infrequently-accessed state. A full description of the fields is given in [Hei98].

In DynPtr, only the home maintains state information on a cache line. Thus, when a remote node misses in its processor cache, the request is immediately forwarded to the home, who keeps track of whether the line is dirty or clean. When a local node misses in its cache, the directory can be consulted on that node.

PTR	Loc	D	P	U	HP	L	R	IO	RP	Dev	SP	HL

FIGURE 12. HeadLink entry for DynPtr. The fields are explained in Table 1.

Field	Full Name	Field Width (bits)	Description
PTR	Ptr	12	Identity of the first sharer in the sharing list
Loc	Local	1	Set if the home (i.e., local) processor is caching the line
D	Dirty	1	Set if the cache line is dirty in a processor's cache or I/O
P	Pending	1	Set if the line is in the midst of a protocol operation
U	Upgrade	1	Set if an upgrade is in progress
HP	HeadPtr	1	Set if the Ptr field contains a valid sharer
L	List	1	Set if there is a sharing list for the cache line (in addition to Ptr)
R	Reclaim	1	Set if the cache line is undergoing pointer reclamation[Hei98]
IO	IO	1	Set if the only valid copy is in the I/O system of some node
RP	RealPtrs	12	Contains the number of sharers in the sharing list
Dev	Device	2	Contains the device number of the IO device requesting this line
SP	StalePtrs	10	Used in conjunction with limited search heuristic[Hei98]
HL	HeadLink	19	Pointer to first element in the linked list of sharers

TABLE 1. Description of the fields of the headLink entry.

3.3.1 Sample Protocol Transactions

To understand the basics of DynPtr, let us consider some sample transactions:

Local read request. When a local node performs a read request, it invokes the PILocalGet handler. If the data is clean, then the request is locally satisfied by memory, and involves no network transactions. If the data is dirty remote, the home forwards the Get request to the remote owner. The remote node invokes the NIRemoteGet handler, which sends the Put data reply to the home. The home runs the NILocalPut handler to send the data to the processor cache. The sequence of actions for the dirty remote case is depicted in Figure 13a.

Remote read request. When a remote node performs a read request, it invokes the PIRemoteGet handler, and the request is forwarded to the home. The home invokes the NILocalGet handler. If the home has the datum (whether clean or dirty), the home adds the requestor to the sharing list and forwards the datum to the requestor (who then performs an NIRemotePut), as depicted in Figure 13b. NIRemotePut forwards the datum to the processor. If the home does not have the datum, and instead a remote node has the datum dirty,

- *Requestor/Home (R/H) invokes the PILocalGet handler. The data is remote, so a GET message (1) is forwarded to the owner (O).*
- *Upon receipt of the GET message (1), the owner invokes NIRemoteGet. The owner retrieves the data from its cache and sends the "PUT with data" message (2) to the requestor.*
- *Upon receipt of the PUT message (2), the requestor invokes NILocalPut, which puts the data in the processor cache and in memory.*

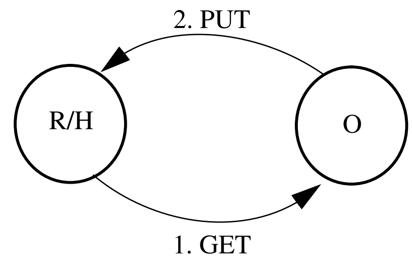


FIGURE 13a. Local read request satisfied by remote owner.

- *Requestor (R) invokes the PIRemoteGet handler. The GET (1) is forwarded to the home (H).*
- *Upon receipt of the GET message (1), the home invokes the NILocalGet handler. The home retrieves the data from either the cache or memory and sends the PUT with data (2) to the requestor.*
- *Upon receipt of PUT message (2), the requestor invokes NIRemotePut, which sends the data to the processor cache.*

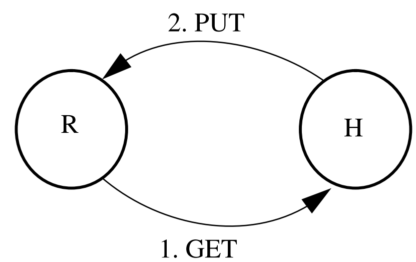


FIGURE 13b. Remote read request satisfied by home.

- *Requestor (R) invokes the PIRemoteGet handler. The GET (1) is forwarded to the home (H).*
- *Upon receipt of GET message (1), the home invokes the NILocalGet handler. The home does not have the data, so the request (2) is forwarded to the owner (O).*
- *Upon receipt of GET message (2), the owner invokes NIRemoteGet. The owner retrieves the data from the cache and sends the "PUT with data" (3a) to the requestor. The owner also sends a sharing writeback message (3b) to the home.*
- *Upon receipt of PUT message (3a), the requestor invokes the NIRemotePut handler, which puts the data in the processor cache. The home invokes NISharingWriteback upon receipt of (3b), adds the requestor to the sharing list, and puts the data in memory.*

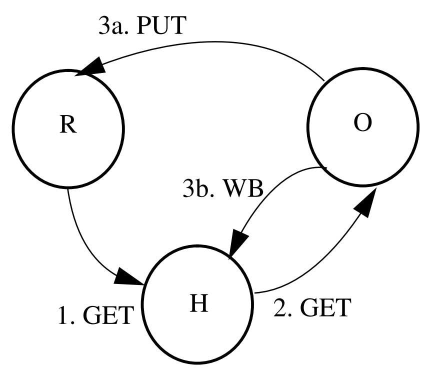


FIGURE 13c. Remote read request satisfied by remote owner.

the home forwards the request to that remote node. The remote node invokes the NIRemoteGet handler and sends both the data reply to the requestor and a sharing writeback message to the home. The sharing writeback signals to the home that it should add the original requestor to the sharing list, and also provides the home with a clean copy of the datum for the backing store. The sharing writeback handler is called NISharingWriteback: since it is only invoked at the home, the Local notation is not needed in the handler name. The sequence of actions described above is depicted in Figure 13c.

- *Requestor/Home (R/H) invokes the PILocalGetX handler. The home sends invalidations (1a,b) to other sharers.*
- *Upon receipt of the INVAL message (1), each sharer invokes the NIInvalidate handler. The cached copy is invalidated, and an invalidation acknowledgment (2a,b) is sent to the home.*
- *Upon receipt of each INVAL_ACK message (2a,b), the home invokes the NIInvalAck handler to count how many acknowledgments have been received. When all acknowledgments have been received, the data is forwarded to the processor cache.*

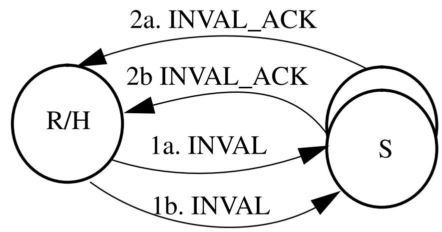


FIGURE 14a. Local write miss, line is shared.

- *Requestor (R) invokes the PIRemoteGetX handler. The data is shared, so the GETX is forwarded (1) to the home (H).*
- *Upon receipt of the GETX (1), the home invokes the NILocalGetX handler, and sends invalidations (2a,b) to all sharers (S).*
- *Each sharer sends an INVAL_ACK message (3a,b) to the home.*
- *When the home receives all INVAL_ACK messages (3a,b), it sends a PUTX message (providing data with ownership) to the requestor (4).*
- *Upon receipt of PUTX (4), the requestor invokes the NIRemotePutX handler, and forwards the data to the processor cache.*

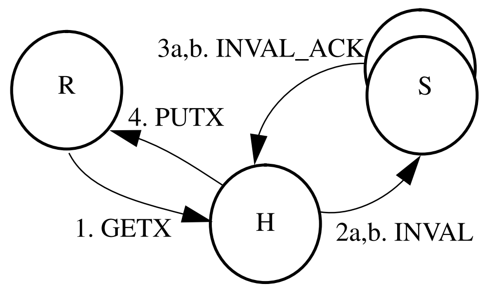


FIGURE 14b. Remote write miss, line is shared.

- *Requestor (R) invokes the PIRemoteGetX handler. The GETX is forwarded (1) to the home (H).*
- *The home invokes NILocalGetX. The data is dirty remote, so it forwards the request (2) to the owner (O).*
- *Upon receipt of the GETX (2), the owner retrieves the data from the cache. It sends the data to the requestor (3a), and sends a forwarding acknowledgment (3b) to the home.*
- *The requestor invokes NIRemotePutX, and forwards the data to the processor cache. The home invokes NIForwardAck, and updates the directory with the new owner (which will be R).*

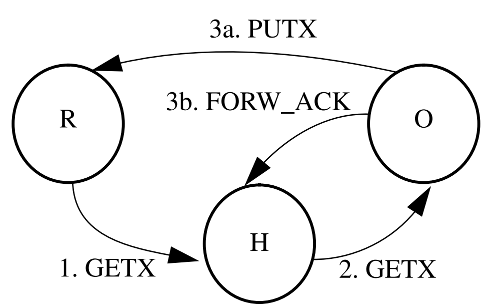


FIGURE 14c. Remote write miss, line is dirty-remote.

Local write request. When a local node performs a write request (PILocalGetX), if the directory indicates that the datum is shared, then invalidations are sent to all sharers. When a sharer receives an invalidation request, it invalidates its cached copy and sends an acknowledgment to the home. Each invalidation acknowledgment invokes the NIInvalAck handler at the home. When the home receives all invalidation

- *Requestor (R), who is also the owner, invokes the PIRemotePutX handler. The data is sent to the home in the REPLACE message (1).*
- *The home invokes the NIWriteback handler, which writes the data to memory and updates the directory to indicate that the requestor is no longer the owner.*

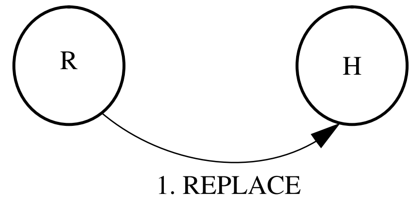


FIGURE 15. Remote writeback.

acknowledgments, it sends the datum to the processor cache with ownership, and indicates the sole ownership in the directory[1] (see Figure 14a). If the datum were originally dirty remote, rather than shared at the home, the request would have been forwarded to the current owner. Because the current owner (running the NIRemoteGetX handler) would have the sole copy of the datum, it would forward the datum directly to the home. The home would then send the data to the processor cache and indicate the change in owner in the directory using the NILocalPutX handler. The picture in this case would be exactly the same as the local read miss satisfied by a remote owner (Figure 13a), with the GetX/PutX handler names appropriately substituted for their Get/Put counterparts.

Remote write request. When a remote node performs a write request (PIRemoteGetX), the request is forwarded to the home. The handler at the home is called NILocalGetX. If the directory indicates that the datum is shared, then invalidations are sent to all sharers. When a sharer receives an invalidation request, it invalidates its cached copy and sends an acknowledgment to the home. When the home receives all invalidation acknowledgments, it sends the datum to the requestor, and indicates the requestor's sole ownership in the directory. The reply handler at the requestor is called NIRemotePutX (see Figure 14b). If the datum were originally dirty, rather than shared at the home, the request would be forwarded to the current owner. Because the current owner (running the NIRemoteGetX handler) would have the sole copy of the datum, it would forward the datum directly to the requestor, sending a forwarding acknowledgment to the home (where the handler is named NIForwardAck). The home would then indicate the change in owner in the directory. These operations are depicted in Figure 14c.

Local Writeback. When a local node needs to replace a dirty datum from its processor cache, the PILocalPutX handler is invoked. Because this node is the home, the backing store is local and the datum is written back to memory immediately. In fact, a speculative memory write can be performed, hiding the protocol handler latency under the latency of the memory reference latency.

Remote Writeback. When a node needs to replace remote data, it invokes the PIRemotePutX handler, which immediately forwards the data to the home. The home, guaranteed to have backing storage for the

1. Sending data to the requestor only after all invalidation acknowledgments have been received is required in order to maintain sequential consistency, which IRIX requires for correctness. Using relaxed memory models would allow data to be returned to the requestor before all invalidation acknowledgments have been received.

line, accepts the line and writes it to memory using the NIWriteback handler. The replacement message from the requestor is not acknowledged, since the replacement is guaranteed to occur at the home. This situation is depicted in Figure 15.

3.3.2 Characterization in the design space

Where is data replicated/migrated? In DynPtr the coherence protocol replicates data in the processor caches, and performs no migration. Home is the backing store for data: if a datum is shared by other nodes, then the home is guaranteed to have a copy of the datum, and as a result, requests for clean data can always be satisfied by the home. If a datum is dirty, then only the owner node contains a copy of that cache line, and only the owner node can satisfy requests for that line. Because a remote node is queried for a line only if it has that line dirty in its processor cache, MAGIC does not store any state on remote lines in DynPtr. Instead, the query is forwarded from the remote MAGIC chip directly to the remote processor, which checks its cache tags to see if the line is actually resident.

Who decides when to replicate/migrate? The cache-coherence protocol is responsible for replicating the data. Whenever a cache line is requested, the appropriate line is located and copied into the requestor's cache. The replication is eager: as soon as the line is requested, it is placed in the appropriate cache. On a write, all other copies of the datum are invalidated.

What happens when a conflict occurs? Because replication occurs only in the processor caches, the only conflicts that occur are cache conflicts. If the line currently in the cache is clean, then a replacement hint is sent to the home, and the home removes the node from the sharing list. Such a message can always be sunk at the home, since it requires merely updating state, rather than sending out additional messages. If the line currently in the cache is dirty, then it must be the only copy of that line remaining in the system. A replacement/writeback message containing the dirty line is sent to the home. The data is guaranteed to be sunk at the home and is written into memory at the home, since the home is the backing store and therefore is guaranteed to have space in memory for the line.

What is the resulting view of memory? Part of memory is used for backing store for PP protocol code/data, including the directory information. The rest of memory is allocatable by the operating system for application code/data.

What are the potential benefits over base cache coherence? This protocol is base cache-coherence. While it is more complex than a full-map directory-based cache-coherence scheme, it should be scalable to larger numbers of processors than a full-map scheme. This scalability issue is not explored in this thesis: for further information the reader is referred to [HS+99][Hei98].

What are the challenges of building this scheme vs. base cache coherence? This protocol is base cache-coherence, but it is more complex than a full-map scheme, primarily because a linked-list is more difficult to maintain than a bitvector. For example, in a bitvector design, to add a node to a sharing list requires simply setting its bit, and removing a node from the sharing list requires simply resetting its bit and sending the invalidation message to the sharer. In DynPtr, adding a node to a sharing list requires allocating a pointer from the pointer free list and then adding that pointer to the existing list. Removing a node requires traversing the list until the appropriate pointer is found, removing that pointer from the list (and sending an invalidation to that sharer), and adding the removed pointer to the free list of available pointers. The resulting handlers are more complex than in a full-map scheme, and replacement hints are required to ensure that lists do not grow too large. The result is that DynPtr adds about 1400 lines of code to a full-map protocol. While this is about 17% of the total DynPtr code size of 8400 lines, the advantage is that DynPtr can keep precise sharing information at larger numbers of processors than a full-map scheme. At small numbers of processors, DynPtr is typically slightly slower than a full-map design, but at larger numbers of processors, the full-map protocol must turn coarse. The extra invalidations induced by the resulting imprecise sharing information causes the full-map scheme to perform worse than DynPtr even at moderate numbers of processors[Hei98].

3.3.3 Static Protocol Complexity

The core DynPtr cache-coherence protocol consists of 65 handlers and 8400 lines of code. These numbers do not include uncached operations and performance monitor handlers since they were not discussed above. These handlers are broken down into 17 PI handlers, 41 NI handlers, 2 IO handlers, and 5 SWQ handlers. PI and IO handlers are generally used for initiating requests, while NI handlers are needed for receiving requests, forwarding requests, receiving replies and generating auxiliary messages like invalidations, as described above.

As mentioned earlier in Section 3.1, two important metrics for evaluating handlers are latency and occupancy. Because MAGIC can issue speculative memory operations, much of the handler cost in DynPtr is hidden under the time to retrieve data from memory. If the latency or occupancy is less than the memory access latency, the cost of the handler execution is completely hidden.

As a baseline against which we can compare other protocols, the uncontended latencies and occupancies for some key handlers in DynPtr are shown in Table 2. These numbers assume that MAGIC runs at 100MHz, and are given in terms of MAGIC cycles. These costs do not include the cost of sending interventions to the processor in order to retrieve data from the processor cache, and assume all directory accesses are cache hits. When the PP sends an intervention to the processor in order to retrieve data, the latency to query the cache and receive the reply (including data) from the processor is approximately 30 cycles. The PP can perform other work while waiting for the result of the intervention, overlapping other useful work with the intervention wait time. For directory accesses that are cache misses, the latency until the first word is returned from memory is 14 MAGIC cycles in the absence of contention.

Handler	Operation	Latency	Handler Occupancy
PILocalGet	request satisfied locally	9	11
	forward request to remote owner	13	14
PIRemoteGet	forward request to home	3	4
NILocalGet	forward data in memory to requestor	10	14
	forward data in cache to requestor (does not include intervention time)	32	33
	forward request to remote owner	15	16
NIRemoteGet	retrieve data from cache and send data to requestor (intervention not included)	10	14
NILocalPut	send data to processor (latency), update directory (occupancy)	1	11
NIRemotePut	forward data to processor	1	2
PILocalGetX	request satisfied locally (no sharers)	11	13
	forward request to remote owner	14	15
PIRemoteGetX	forward request to home	3	4
NILocalGetX	forward data in memory to requestor (no sharers)	12	20
	forward data in cache to requestor	30	34
	forward request to remote owner	15	19
NIRemoteGetX	retrieve data from cache, send data to requestor, and send ack to home (intervention not included)	17	23
NILocalPutX	send data to processor (latency), update directory (occupancy)	1	11
NIRemotePutX	forward data to processor	1	2

TABLE 2. Latencies and occupancies of common protocol handlers in DynPtr. Latency to access memory or access the processor cache (through a processor intervention) is not included. The latency and occupancy values are given in terms of MAGIC 100 MHz cycles**.**

For most remote handlers, the latency and occupancy are very similar, as Table 2 shows. In these handlers, there is no directory lookup required, and the difference between latency and occupancy is usually due to sending some sort of acknowledgment to the home, or simply invoking the instruction to exit a handler. For example, PIRemoteGet simply takes the processor request, sets the outgoing message type, sends the request, and exits. These operations are inherently serial, so it takes 3 cycles to send the message and another cycle to exit the handler. NIRemoteGet is slightly more expensive: an incoming message must be translated into an intervention to be sent to the processor. MAGIC must then prepare an outgoing message and check the result of the intervention to see if the processor actually had the data. After a successful intervention, MAGIC checks to see if the home is also the requestor. If not, then the requestor gets the data (contributing to latency) and the home gets an acknowledgment (contributing to occupancy). If the home is the requestor, then the home gets one message that acts as both a data reply and an acknowledgment. Formatting messages typically takes a few cycles: one cycle for each field that must be set. These small costs add up to the latencies and occupancies shown above.

Local requests are more complex because the requesting node contains directory state that must be consulted. For example, in PILocalGet, the directory address must be calculated, and then the directory must be accessed. These actions take about 5 cycles: 2 cycles to form the directory address, and 3 cycles to perform the load since the directory data cache is off-chip. After performing these actions, MAGIC must then check pending and dirty bits to see either if any other accesses are in progress for the line, or if it is dirty on some other node. If none of these conditions are true, then the data from memory can be used. In PILocalGet, a speculative memory operation is used to overlap data retrieval from memory with protocol processing, so the data is ready by the time all these checks have completed. In a handler like NILocalPut, first the data is sent to the processor, and then directory updates (like setting a bit to indicate the home is a sharer, resetting the dirty bit, resetting the pending bit, and writing back the modified directory to memory) are performed. These updates consume an additional 10 MAGIC cycles after the datum has been forwarded to the processor, and are overlapped with a speculative memory write of the datum to its local backing store.

3.4 Coherent-caches plus Remote Access Cache (RAC)

While DynPtr is conceptually simple, if the application data set overflows the processor caches, performance can suffer because data will need to be fetched from the home. The caches simply might not be big enough to provide sufficient data locality and replication. The Remote Access Cache (RAC) protocol attempts to improve data locality by using part of main memory as a cache, supplementing the processor caches. Conceptually, the RAC is a tertiary cache for remotely-allocated data. A number of commercial machines and academic prototypes incorporate some form of RAC in their designs, including DASH[LLJ+92], Data General NUMALiiNE[Dat97], Sequent STiNG[LoC96], Typhoon[RLW94], and HP Exemplar[BrA97], although in DASH the motivation was different from here: the RAC was used to interface an SMP snooping bus with a directory-based cache-coherence protocol.

The RAC protocol requires an underlying directory-based cache-coherence protocol to keep track of data in the processor caches as well as data in the RACs. In our implementation, RAC is built on top of DynPtr, and extends its functionality. A portion of memory is statically-partitioned as a cache for remotely-allocated data (hence the name Remote Access Cache), as shown in Figure 16. Whenever a remote datum is requested, that datum is copied not only into the requestor's processor cache, but also into the RAC portion of its memory. If that line is subsequently evicted from the processor cache, a copy may still remain in the RAC, and the request can still be satisfied without leaving that node. Note that locally-allocated data does not need to go into the RAC: when a local datum is in the processor cache, and is later evicted, because it is locally-allocated, it is by default written back to local memory, and subsequent requests will already be satisfied locally.

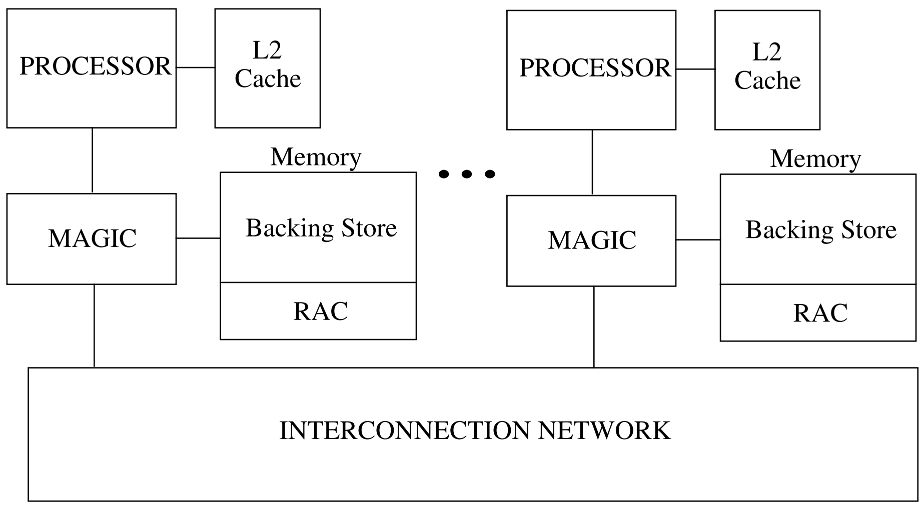


FIGURE 16. System-level organization of RAC design. A portion of local memory on each node is devoted to a cache for remote data. The rest of memory on each node is used for normal backing store purposes. On a processor cache miss, remotely-allocated data is copied both into the RAC as well as the processor L2 cache.

Also note that because the associativity of the RAC and the processor cache may differ, inclusion is not maintained between them, leading to more aggregate space for replication (i.e. replicated data may exist either in the RAC or in the processor cache).

The RAC utilizes the fact that a cache in memory can be much larger than a hardware cache, and that the fraction of memory devoted to RAC can be reconfigured at boot-time. Hardware caches, in contrast, typically cannot be expanded above a certain size that is fixed at system design time. The RAC also caches only remote data, while the hardware cache stores both local and remote data: while this means there are no RAC collisions between local and remote data, if a significant fraction of misses occurs to remote data, the RAC itself may also overflow, potentially causing performance degradation.

3.4.1 Sample Protocol Transactions

Let us now consider some sample transactions for RAC.

Local read request. The initial handler invoked is PILocalGet, as in DynPtr. Because the requestor is already the home, the directory is checked. If the data is in memory, it is returned to the processor. If the data is dirty remote, the request is forwarded to the remote node. The remote node (in handler NIRemoteGet) sends the reply data to the home, and the data is returned to the processor. This is exactly the same sequence of operations as in DynPtr, except that the remote node may have the datum either in its RAC or in its processor cache, rather than just in the processor cache.

Remote read request. The addition of a RAC means that unlike DynPtr, information must be kept at remote nodes: this information is the tags and state for the RAC. Remote references in RAC (in the PIRemoteGet handler) must first check the tags/state in the RAC to see if the requested datum is in the RAC. Recall that in DynPtr, the request is immediately forwarded to the home, with no state checks required. If the datum is in the RAC, it is returned to the processor. If not, the request is forwarded to the home. The home runs the NILocalGet handler. If the home has the datum (i.e. it is not dirty remote), it returns the datum to the requestor, and the datum is both sent to the processor and copied into the RAC (in handler NIRemotePut). If the home does not have the datum, then the request is forwarded to the remote node containing the datum. Again, unlike in DynPtr, a remote node might have the datum in either the processor cache or the memory RAC, so both may need to be checked in the NIRemoteGet handler. In any case, the datum is returned to the requestor, with the home sent a sharing writeback so that the requestor can be added to the sharing list and the home can get a clean copy of the datum. The requestor sends the reply data to the processor and also copies the datum into the RAC in the NIRemotePut handler.

Local write request. The same actions occur as for DynPtr, except that a remote node may have the datum in its RAC rather than in its processor cache.

Remote write request. First, the RAC is checked. If the datum is present, and is the only remaining copy in the system, it is immediately returned to the processor. Such a situation might arise if a processor requested a datum, wrote it, and then had the datum evicted from the cache to the RAC. If the datum is not in the RAC, the request is forwarded to the home. From here, the same actions as for DynPtr occur, except for possibly checking the RAC at remote nodes.

One complication that arises on remote requests is that when the reply reaches a node and the datum is to be copied into the RAC, the RAC may have another valid datum at the same location. This datum (if it is the only remaining copy) must be written back to the home before the new datum is copied into the location. The home is guaranteed to sink such requests, since it still serves as backing store, as in DynPtr.

Local Writeback. Because the RAC is used only for remote data, the local writeback case is the same as for DynPtr.

Remote Writeback. When a node needs to replace remote data, it invokes the PIRemotePutX handler. Unlike DynPtr, PIRemotePutX first checks if there is a placeholder for the dirty datum in the RAC. If so, the datum is written back to the appropriate location in the RAC, and is not sent home. If there is no such placeholder, the home is sent a writeback request. The home is guaranteed to sink this request, as in DynPtr.

3.4.2 Characterization in the design space

Where is data replicated/migrated? As in DynPtr, data is replicated, but not migrated, by the coherence protocol. Local data is replicated only in the processor caches. Remote data is replicated both in the processor caches as well as in the RAC portion of memory. If a datum is shared, then the home retains a copy of the datum. If a datum is dirty (residing either in a cache or in a RAC), then only the owning node can satisfy requests for that line.

Who decides when to replicate/migrate? The cache-coherence protocol replicates the data. Whenever a cache-line is requested, the appropriate line is located and copied into both the requestor's RAC (for remote lines) and the requestor's cache. This replication is eager. On a write, all other copies of the datum are invalidated.

What happens when a conflict occurs? Replication of local data occurs only in the cache. A cache conflict for local clean data results in a replacement hint, while a conflict for local dirty data generates a writeback to local memory. Replication of remote data occurs both in the processor cache and in the RAC. A cache conflict for remote clean data generates a replacement hint. If the RAC also has a copy of that data, then it sinks the replacement hint. If the RAC does not have a copy of that data, then the replacement hint is forwarded to the home. A cache conflict for remote dirty data generates a writeback request to the RAC. If the RAC has a placeholder for that data, it sinks the writeback. If not, the writeback is forwarded to the home. A RAC conflict for clean data generates a replacement hint to the home, while a RAC conflict for dirty data generates a writeback to the home. Note that strict inclusion between cache and RAC is not maintained, to provide more aggregate room for replication.

What is the resulting view of memory? Part of memory is used for backing store for PP protocol code/data, and for the RAC tags/state. The rest of memory is split between that allocatable by the OS for application code/data, and that partitioned for the RAC.

What are the potential benefits over base cache coherence? RAC attempts to eliminate capacity misses by replicating data into local node memory. The more RAC hits there are, the fewer costly remote references there are, and the smaller the resulting memory latency is relative to DynPtr. Each remote miss satisfied by the RAC also reduces the number of requests to the home, alleviating contention in the network and in the PP at the home. In addition, the requestor will also run fewer handlers since on RAC hits there is no need for an NIRemotePut/NIRemotePutX handler (the request is instead satisfied locally in the PIRemoteGet/PIRemoteGetX handler). Indirectly, the memory bandwidth requirements of the home will be reduced since it will need to make fewer references, although the memory bandwidth requirements at the requestor will increase due to the RAC queries.

What are the challenges in building this scheme vs. base cache coherence? While the RAC offers the potential of improved locality, the improvements come at the cost of increased complexity. The protocol is built upon DynPtr, so part of the complexity involves the sharing-list manipulations described earlier. Because only remote data goes into the RAC, local misses are treated the same in RAC as in DynPtr, so local handlers are identical between the two protocols. The bulk of the complexity increase involves servicing remote misses. We will first discuss the changes required relative to DynPtr qualitatively, addressing the

AmP	I	AmT	AmS	Offset

FIGURE 17. Data structure for RAC tag and state information. The fields are explained in Table 3**.**

three main areas in which changes are required: data structure organization, processor remote requests, and network remote requests. We will then describe the quantitative effects of each of these sets of changes.

1. **Data structure organization**. The RAC uses the same directory organization and headLink data structure that DynPtr uses. In addition, tag and state information is required for memory lines in the RAC portion of memory on a node. This data structure is shown in Figure 17, with a brief description of the fields shown in Table 3. The new information required for the RAC includes the following:

    - *AmPending* bit: This bit is set if there are any operations outstanding on this RAC line. It is similar to the Pending bit in the directory. If a processor submits a request to the RAC which misses, and the RAC line already has a request outstanding on it, then the processor's request is NACKed and retried, to avoid difficulties of storing information for multiple requests to the same RAC line.
    - *Invalidate* bit: This bit is set if an invalidation has been received for this RAC line while a request is already outstanding for this line. When the reply returns, it checks the status of the Invalidate bit. If this bit is set, then the reply data may be invalid. The PP must perform actions similar to the conflict resolution table (CRT) in the PI[Kus97] to determine what to do with the reply data.
    - *Amtag* bits: These bits store the tag. Because our RAC is direct-mapped, the tag is essentially the home node of the data. Because multiple lines from the same node can map to the same entry in an another node's RAC, the tag is used in conjunction with the offset bits to determine whether a tag match has occurred.
    - *Amstat* bits: These bits store the state of the RAC line, described in Table 4.
    - *Offset* bits: These bits store the offset bits of the physical address stored in the RAC, and are used in conjunction with the tag to uniquely identify the address in the RAC.

2. **Processor remote request changes**. Processor interface (PI) remote requests must check if data is in the RAC before being forwarded to the home. In DynPtr, remote requests were automatically forwarded to the home. The extra RAC check time adds overhead, but it is hoped that requests will be satisfied in the RAC, eliminating the need to go to the home as frequently, and subsequently improving performance. Replies to PI requests must place the data in the RAC as well as send the data to the cache. Not only may putting data in the processor cache result in a cache conflict, but placing data in the RAC may result in a RAC conflict. Writing the data to the RAC and its associated conflict processing adds PP code overhead that is not present in DynPtr. Another cost is that storing state at the remote node will hurt PP data cache performance at the remote node, since it will have to access RAC information in situations where previously it did not have to access any state.

Field	Full Name	Field Width (bits)	Description
AmP	AmPending	1	Set if any operations are pending on this RAC line
I	Invalidate	1	Set if an invalidation has been received for this RAC line while a request is outstanding for this line
AmT	Amtag	8	Stores tag information
AmS	Amstat	3	Stores state information
Offset	Offset	29	Stores offset portion of address stored in RAC

TABLE 3. Description of the fields of the RAC tag and state data structure.

RAC State	Description
SHAR	Line is shared and only in the RAC
SHARS	Line is shared and in both the RAC and processor cache
EXCL	The RAC has the only copy of the line
EXCLX	Processor cache has a dirty copy of the line and the RAC has a placeholder for the line
INVL	RAC entry is empty

TABLE 4. Description of RAC states.

3. **Network remote request changes**. Incoming requests from other nodes (Network Interface, or NI, requests) must be handled differently from DynPtr, if the node receiving the request is not the home for the data. In DynPtr, a remote node receives a request only if it contains the data dirty in its cache, so the cache is immediately queried for the data; in the RAC protocol, a remote node may contain the data either in its cache or in its RAC, so both may have to be queried for the data. Again, remote references that consult RAC state will see worse PP cache performance as a result, since the cache now stores both directory state and RAC state. Also, replies will forward data not just to the processor, but also to the RAC, requiring updates to the RAC state.

3.4.3 Static Protocol Complexity

While RAC adds functionality to DynPtr, simplifying the design of the RAC can lead to very small static overheads over DynPtr. Our RAC design adds only 1 handler to DynPtr, giving a total of 66 handlers. This additional handler is used in notifying the home when a RAC downgrades its copy of a line from exclusive to shared. This functionality is required because our RAC is direct-mapped but the processor cache is two-way set-associative. For example, consider a dirty line that has been evicted from the processor cache, but is written back to the RAC (rather than the home). The line might be accessed later, so that a copy exists both in the processor cache and in the RAC. Because this was a dirty line, the home will forward all requests to

Handler	Operation	Latency	Handler Occupancy
PILocalGet	request satisfied locally	10 (+1)	12 (+1)
	forward request to remote owner	14 (+1)	15 (+1)
PIRemoteGet	request satisfied from RAC	16 (n/a)	17 (n/a)
	forward request to home (RAC miss)	20 (+17)	31 (+17)
NILocalGet	forward data in memory to requestor	11 (+1)	15 (+1)
	forward data in cache to requestor (does not include intervention time)	33 (+1)	34 (+1)
	forward request to remote owner	16 (+1)	17 (+1)
NIRemoteGet	retrieve data from cache and send data to requestor (intervention not included)	16 (+6)	26 (+12)
	retrieve data from RAC and send data to requestor	25 (n/a)	38 (n/a)
NILocalPut	send data to processor (latency), update directory (occupancy)	1	11
NIRemotePut	forward data to processor and RAC	1	40 (+38)
PILocalGetX	request satisfied locally (no sharers)	11	13
	forward request to remote owner	14	15
PIRemoteGetX	request satisfied from RAC	17 (n/a)	18 (n/a)
	forward request to home (RAC miss)	20 (+17)	31 (+27)
NILocalGetX	forward data in memory to requestor (no sharers)	13 (+1)	20
	forward data in cache to requestor	31 (+1)	35 (+1)
	forward request to remote owner	16 (+1)	20 (+1)
NIRemoteGetX	retrieve data from cache, send data to requestor, and send ack to home (intervention not included)	19 (+2)	33 (+10)
	retrieve data from RAC and send data to requestor	27 (n/a)	41 (n/a)
NILocalPutX	send data to processor (latency), update directory (occupancy)	1	11
NIRemotePutX	forward data to processor and RAC	1	38 (+36)

TABLE 5. Latencies and occupancies of common protocol handlers in RAC. The differences, if any, between RAC and DynPtr latency/occupancy values are shown in parentheses next to the corresponding latency/occupancy. n/a means not applicable, and is used for cases that exist in RAC but not in DynPtr. Latency to access memory or access the processor cache (through a processor intervention) is not included. The latency and occupancy values are given in terms of MAGIC 100 MHz cycles. When a datum is sent to both the RAC and the processor, the latency value includes only the time required to forward the datum to the processor.

this line to this node. If this node evicts the line from the RAC, then it will get written back to the home, but this node's processor cache will still have a copy, so special handlers are needed to indicate this situation.

Of the remaining handlers that are shared between RAC and DynPtr, only 5 of the 17 PI handlers must be changed: these are all remote handlers, reflecting changes required for accessing the RAC on remote references. Local handlers are the same as DynPtr. Of the NI handlers, not including the 1 extra handler mentioned above, only 9 of the 41 NI handlers need to be changed for RAC. Again, these are all remote handlers. One of the two IO handlers must changed (a remote IO GetX handler) to incorporate checking the RAC before forwarding a request to the home, and no SWQ handlers differ between RAC and DynPtr. In all, 15 of the 65 DynPtr handlers must be changed to accommodate RAC. While DynPtr consumes roughly 8400 lines of code, RAC is roughly 8900 lines of code, adding about 500 lines to DynPtr. The changes are non-trivial, but small in quantity.

The static differences between DynPtr and RAC lead to changes in the latencies and occupancies of the modified handlers, as seen in Table 5. Next to each latency and occupancy value, in parentheses, is the difference in latency/occupancy between RAC and DynPtr. Latencies for PI and NI local handlers are practically identical to those of DynPtr, since those handlers are unchanged relative to DynPtr.[2] Remote handlers are higher latency and occupancy than DynPtr because of the RAC tag and state checking overhead. The differences can be rather dramatic in some cases. For example, consider NIRemotePut. In DynPtr, this handler simply forwards data to the processor and exits. In RAC, however, the RAC state must be accessed and updated. While the bulk of replacement processing in RAC is performed when a request is initially generated, a small amount is still performed when a reply returns: this processing is required because lines in the processor cache may be written back to the RAC while the current request is outstanding. Performing the necessary RAC checks and updates leads to the increased occupancy. The major costs involve accessing the tags (9 cycles), checking if the replacement processing is required (4 cycles), acquiring the resources and setting up the message (17 cycles), and modifying the tags and writing the data to the AM (8 cycles). These costs are similar for NIRemotePutX, which must do the same checks. In PIRemoteGet, while the RAC checking adds quite a bit of overhead relative to DynPtr, if a RAC hit occurs, then no costly network accesses are required, and no processing at the home is required, reducing the overall latency of the reference. In handlers that may need to check both the RAC and the cache, requests must be sent to both the RAC and the cache. Because interventions are high latency, the intervention to the processor is sent first, and the RAC is checked while the intervention is pending. To properly deallocate resources, the intervention must be allowed to complete even if the RAC (rather than the cache) has the line, adding latency to the handler even in the event of a RAC hit.

Overall, the latencies are certainly higher for RAC than for DynPtr. In the results chapter we will discuss whether the locality enhancements outweigh the increased latencies. In noting these latency differences, one point to consider is whether the MAGIC PP's ISA is suitable for implementing a RAC. For MAGIC to per-

2. Local handler occupancy may differ slightly from DynPtr because of an optimization in directory addressing that can be performed in DynPtr. This optimization is possible because in DynPtr, only home nodes access directory information, while in RAC the unused directory entries are used to store tag information, which can be accessed by remote nodes. The optimization shaves 1 cycle off directory access time in DynPtr relative to RAC.

form operations more efficiently would require very specific primitives. For example, accessing the tags takes 9 cycles, of which 6 are required to form the address of the tag data structure and 3 are needed to perform the load. To do this operation more efficiently would require a hardware lookup table in MAGIC. Unless such a table were valuable for other protocols, it would be difficult to justify inclusion of such a hardware structure. Optimizing MAGIC for all possible protocols simply cannot be done. Another possible source of optimization is in setting up message registers. Properly setting the values of registers involves bit shifts and insertions, which can be used for all protocols and already involves special MAGIC PP instructions. Next, consider resource acquisition. Resource acquisition typically involves checking the status of various queues in MAGIC. There are already special commands for such checks which usually incur 3 to 4 cycles of latency. Finally, the actual modification of tags and AM data involves register manipulations and loads and stores to memory. There are special cache-line sized loads and stores to memory for AM data, namely lblock and sblock. The modification of tags involves bitfield insertions and shifts, for which special instructions already exist.

As the above discussion demonstrates, the MAGIC PP ISA is not really limited by lack of special instructions. Instead, MAGIC could potentially benefit if more operations could be done in parallel. For example, allowing the acquisition of resources while other checks are going on would improve performance. To implement such parallelism, support for more speculative execution would be required. However, protocol writing might then be more difficult, since it would involve specifying speculative threads of execution. The design of MAGIC instead focuses on simplicity to achieve a higher clock rate and reduce the burden on the protocol programmer.

3.5 Cache-Only Memory Architecture (COMA)

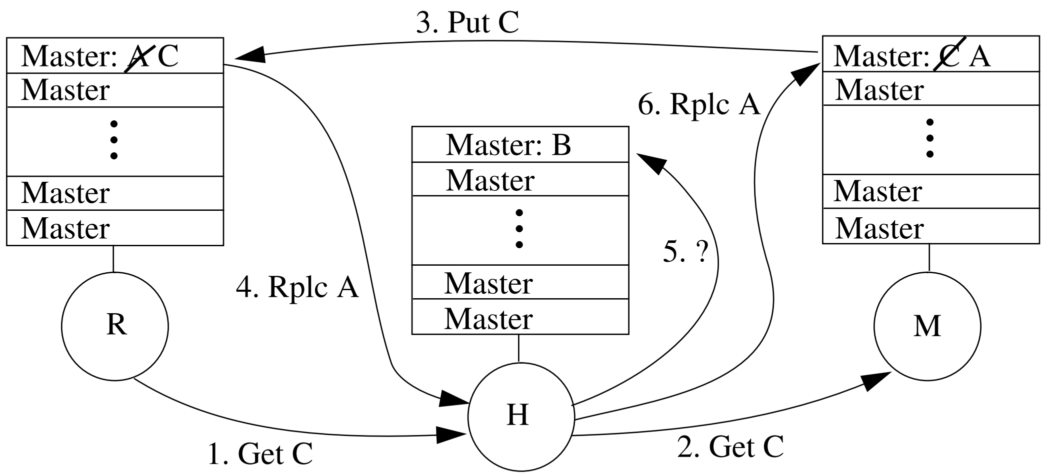
RAC attempts to improve on DynPtr by providing a tertiary cache for remote data in a portion of memory. If the problem data set overflows the processor, and if the remote data fit in the RAC, then RAC can provide gains over DynPtr by converting remote misses into local hits. But suppose a poor initial placement results in a majority of the data for a given processor's computation being allocated remotely. Most of that processor's requests will be remote requests, and the RAC will quickly fill up, causing conflicts and RAC misses and potentially degrading performance. In effect, for good performance, the static partitioning used by RAC relies on proper initial allocation of some amount of data.

A Cache-Only Memory Architecture (COMA) attempts to eliminate any such dependence on initial page placement. In COMA, all of memory is treated as a cache. Similar to the RAC, a miss on a line causes that line to be replicated both into the cache and into memory. Unlike RAC, there is no fixed node where space is guaranteed to be allocated for a memory block (i.e., there is no static backing store for data). Instead the hardware is responsible for tracking all outstanding copies of a line. When replacements occur for that line, the hardware tracks the remaining copies, making sure that one copy of each memory lines exists somewhere in the system. Every line in memory is tagged, so any line in memory can be used to store locally-allocated or remotely-allocated data, unlike the RAC, in which only a specified portion of memory can hold remote lines. Because any line in memory can hold remote or local data, the memory can dynamically respond to the access patterns in the applications. If a given node references only remote data, then its entire memory can be filled with remote data. Any datum that had originally been allocated on that node would then need to reside elsewhere, presumably on the node on which it is accessed. By allowing such dynamic partitioning of memory, COMA avoids some of the problems of the static partitioning approach used in the RAC protocol, though at a cost of increased complexity.

The proposed advantage of COMA is that data movement happens without the user or OS intervention. Thus, a commodity OS and a commodity application can run essentially unmodified on a COMA system. There have been a number of COMA protocols proposed to date, including Hierarchical COMA (COMA-H) [HLH92][FBR93], Flat COMA (COMA-F) [SJG92], and Simple COMA [HSL94]: the implementation we describe here is based on the COMA-F protocol introduced by Stenstrom and Joe [SJG92][JoH94]. Because this protocol is the first implementation of a real COMA-F design (as opposed to a high-level behavioral model of a protocol), we will discuss our implementation in some detail.

In COMA, the entire per-node main memory is converted into a large tertiary cache called an attraction memory, or AM, because lines are "attracted" to the memories in which they will reside. The home is not a static backing store for data: instead, it stores only directory information used to track the outstanding copies of a line. When a datum is referenced, it is replicated in the requestor's AM, and one replica is considered the backing store. This line need not reside at the home of the datum. The replica that is the backing store is referred to as the *master* copy. If there is a dirty copy of a line, then that copy is the master (or owner) by default, since no other copy can exist in the system. Unlike in RAC, in which eviction of any line from the RAC requires simply a writeback to the home node, in COMA the home is not a guaranteed backing store. Instead, an evicted line is sent to the home, and if the line is the master copy, then the home is responsible for finding another node to serve as the master, so that the line is not lost. If the line is not a master copy, then it can simply be discarded by the home.

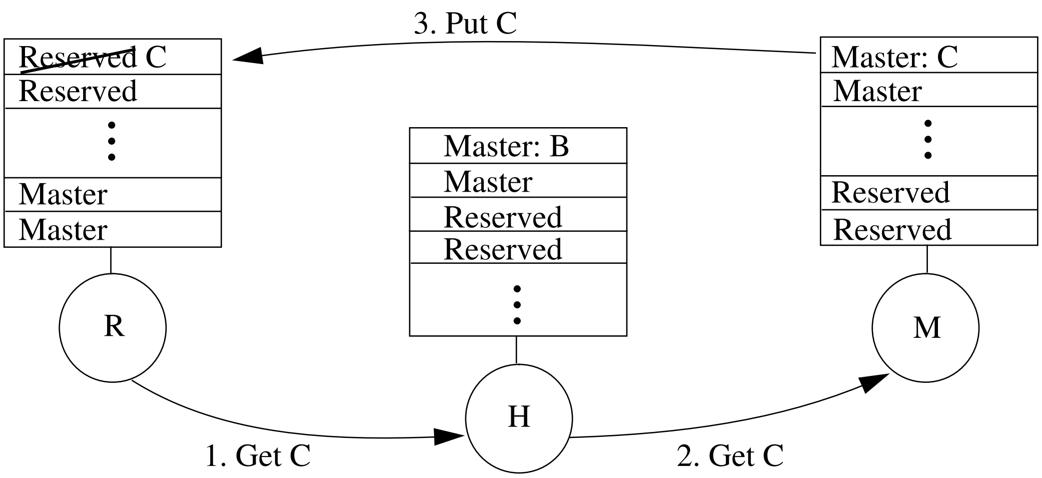
There are four main differences between RAC and COMA. First, only a portion of memory is used as a cache in RAC, while all of memory is a cache in COMA. Second, a RAC contains only remote data, while an AM can contain remotely-allocated data as well as locally-allocated data. Third, RAC still retains a static backing store at the home of the line, while in COMA, the home is merely a static placeholder for the directory data. COMA removes the notion of a static backing store for data in order to allow the movement of data anywhere it is needed in the system, unhindered by any static partitioning of memory. The final main difference between RAC and COMA is that COMA does not need to keep unnecessary copies of data around in the system. If a node never accesses a line, and happens to be the home for a line, that node is not required to keep the line in its local memory.



- *The requestor (R) originally has line A in its AM, and wants line C. It sends a read request (1) to the home of C (H).*
- *The home does not have the data, so it forwards the request (2) to the current master (M).*
- *The master sends data line C (3) to the requestor. The acknowledgment to the home is not shown for simplicity.*
- *The requestor receives the data (3), but is master to line A in its AM, so it sends a replacement request (4) for line A to the home of A (also H).*
- *The home of C (H) already has a master in the slot that line A would occupy (5), so it forwards the request (6) to the old master of C (M).*
- *The old master of C (M) has a free slot, since it just surrendered line C, so it sinks line A. The acknowledgment to the home is not shown for simplicity.*

FIGURE 18. Migration in COMA when all of memory contains master copies.

Replication occurs in COMA by copying a requested line from the master node into the requestor's AM and cache. Note that a copy implies that the backing store (i.e. master) exists elsewhere in the system, just as caches represent "extra" storage space with memory as backing store. Migration in COMA occurs by moving a master (i.e. the backing store) to a requesting node. For the best performance it is typically desirable to allow both replication in the AM as well as migration. But imagine if every memory line were a master copy: in that case, every reply datum would displace a master copy in its AM. Not only would this cause significant network traffic to find new spots for the masters, but replication in the AM would be impossible, because no "extra" slots would exist for copies of data. This situation is exhibited in Figure 18. Replication would only be possible in the caches. In order to allow replication both in the caches and the AM, a certain fraction of memory is *reserved* from allocation by the OS[JoH94]. A line allocated by the OS is by definition a master copy because it is initially the only copy of that line in the system. If every page in memory (and therefore every line in memory) were available for the OS, then every line could be a master copy, and as described above, we would have no replication potential except in the processor caches. In contrast, reserved memory is not allocatable by the OS. These reserved lines can thus store master copies from other nodes, or simply store replicated copies of data. If a reserved slot stores a master copy, then the node from whose



- *A portion of memory on each node is reserved from allocation by the OS. These lines are initially empty.*
- *The requestor (R) wants line C, so it sends a read request (1) to the home of C (H).*
- *The home does not have the data, so it forwards the request (2) to the master of C (M).*
- *The master sends line C to the requestor (3).*
- *The requestor's AM entry had been reserved, so it can be filled with line C without evicting any master copies. The requestor's AM may now contain a copy of line C, while the master may still be node M.*

FIGURE 19. Reserved memory and replication potential in COMA.

memory the master has been retrieved has an empty slot at that offset in its AM. This situation is depicted in Figure 19. By reserving memory and not allowing some of it to be allocated by the OS, a fraction of memory will thus not contain a master copy, and a reply datum has a probability of not displacing a master copy and instead residing in this "extra" space. In addition, the coherence protocol can now choose to merely replicate a datum rather than migrate it as long as there is space for it. In general, the amount of reserved memory can be dynamically determined, rather than statically partitioned. As a result, COMA can potentially provide more flexibility in terms of deciding when to migrate or replicate.

3.5.1 Sample Protocol Transactions

Let us now consider some sample transactions for COMA.

Local read request. The initial handler invoked is PILocalGet, as in DynPtr/RAC. Because all of memory is a cache, the tags and state for this memory line must be checked to see if the data is in the AM. If so, the line is immediately returned to the processor. If not, the directory is consulted in order to determine the current master. The request is then forwarded to the master. The master invokes the NIRemoteGet handler. The state of the line is checked, and either the cache or AM is queried for the line, which is then returned to the requestor. The requestor performs the NILocalPut, which returns the datum to the processor and copies the datum into the AM. Note that when the NILocalPut is invoked, the AM may already contain a master copy

- *Requestor/Home of line A (R/H_A) invokes PILocalGet. The GET is forwarded (1) to the master of line A (M_A).*
- *Upon receipt of the GET message (1), the master of line A invokes the NIRemoteGet handler. The master retrieves the data from either the cache or AM and sends the data (2) to the requestor.*
- *Upon receipt of the PUT (2), the requestor invokes NILocalPut. Since the requestor is master to line B, which is at the offset that line A will go to in the AM, the requestor must send a replacement message for B (3) to the home of B (H_B).*
- *The home of B invokes the NIRplc handler, sends an acknowledgment (4a) to the requestor, and also sends the line (4b) to another node (M_B) to request that it become the new master for line B.*
- *The queried node (M_B) invokes the NIPutRplc handler, agrees to store line B, and sends an acknowledgment (5) to the home of B.*

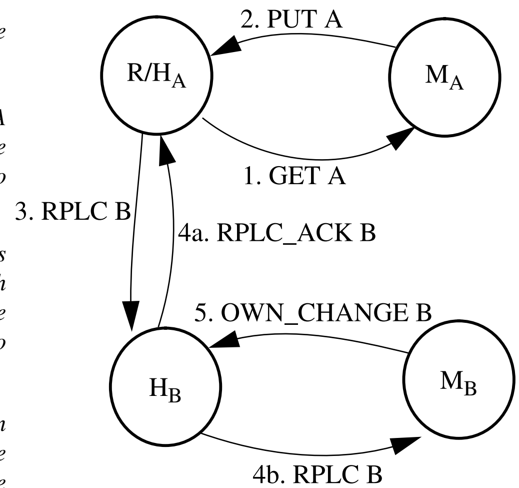


FIGURE 20a. Local read request satisfied by remote master.

- *Requestor of line A (R) invokes PIRemoteGet. The GET is forwarded (1) to the home of line A (H_A).*
- *The home of A does not have the data, and forwards the GET (2) to the master of A (M_A).*
- *The master receives the GET, and retrieves the data from either the AM or the cache. It sends the data to the requestor (3a) and sends an acknowledgment (3b) to the home.*
- *The home of A receives the acknowledgment and invokes the NIReqAck handler. It changes the master to the requestor, and adds the previous master to the sharing list.*
- *At the requestor node, line A conflicts in the AM with line B. The requestor sends a replacement (4) to the home of B (H_B)*
- *The home of B sends an acknowledgment to the requestor (5a) and sends line B (5b) to another node (M_B) to request that M_B become the new master of B.*
- *The queried node (M_B), agrees to store line B, and sends an acknowledgment (6) to the home of B.*

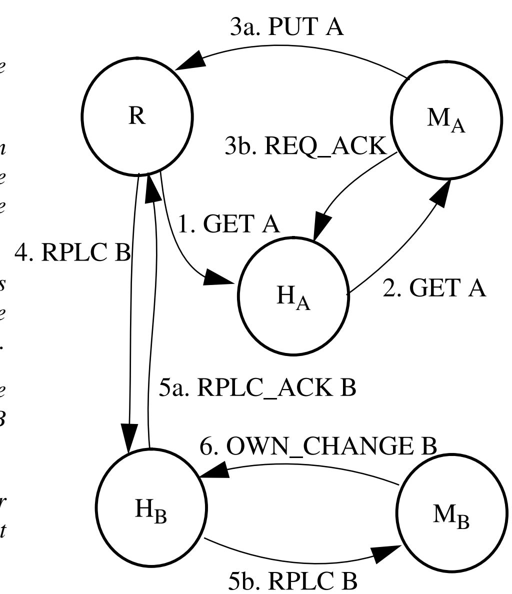


FIGURE 20b. Remote read request satisfied by remote master.

at the offset where the reply data will be placed. In this event, conflict/replacement processing will have to occur. Figure 20a depicts these events. Note that conflict processing never happens in the PP in either RAC or DynPtr for local data: the only local data conflicts that can occur in those protocols are conflicts in the processor cache, which are handled by the processor's internal cache controller.

Remote read request. Similar to RAC, tag and state information is kept at remote nodes. Remote references in COMA invoke PIRemoteGet, which must first check the tags/state to see if the requested datum is on-node. If so, the datum is returned to the processor. If not, the request is forwarded to the home. The home runs the NILocalGet handler. If the home has the datum, the home returns the datum to the requestor, and the datum is both sent to the processor and copied into the AM (in handler NIRemotePut). Notice that, unlike DynPtr and RAC, the datum can be in a shared state, yet the home might still lack the datum: this is an artifact of the lack of a static backing store for data. If the home does not have the datum, then the request is forwarded to the remote node containing the datum (the master). Note that, similar to RAC, a remote node might have the datum in either the processor cache or the memory (the AM), so both may need to be checked in the NIRemoteGet handler. In any case, the datum is returned to the requestor, with the home sent a sharing writeback so that the requestor can become the master and the old master can be added to the sharing list. The requestor is made the master under the assumption that it is more likely to reference the line repeatedly in the future than the old master. If the home has room, it sinks the writeback datum. If not, it ignores the datum. By sinking the writeback datum, the home can satisfy future read requests without having to consult the master, an optimization over consulting the master on each request. The original COMA-F design[SJG92] specified that all requests must go to the master to retrieve a datum, in order to simplify the process of tracking the master. However, forcing read requests to go to the master even if the home has a copy is not necessary and increases response latency, so we allow our protocol to perform this optimization. The requestor sends the reply data to the processor and copies the datum into the AM in the NIRemotePut handler, performing any necessary replacement processing. These events are depicted in Figure 20b.

Local write request[3]. When a local node performs a write request (PILocalGetX), if the directory indicates that the datum is shared, invalidations are sent to all sharers. Unlike in DynPtr and RAC, in which a shared directory state indicates a copy exists at the home, in COMA no such guarantee exists. Instead, the home may not have the datum. In order to simplify the protocol, all write misses must go to the master–this eliminates a number of transient conditions in which two nodes might be master at the same time. If the home is the master, it waits for all invalidation acks to return, and then returns the data to the processor, similar to DynPtr as shown in Figure 14a. If the home is not the master, it forwards the request to the master, as shown in Figure 21a. The master invokes an NIRemoteGetX, and returns the data to the home. After the home receives both the data and all the invalidation acks, it returns the data to the processor and copies the data into the AM, performing the AM conflict resolution at that time. If the master has the only copy of the line it

3. We again assume a sequentially-consistent memory model.

- *The requestor/home (R/H), sends a GETX request (1a) to the master (M), and issues invalidations (1b,c) to all current sharers (S).*
- *The master retrieves the data from the AM or cache and sends the data to the home (2a).*
- *Each sharer sends an invalidation acknowledgment to the home (2b,c).*
- *When the home receives the data (2a), it invokes the NIReqXAck handler, and stores the data until all the invalidation acknowledgments have been received.*
- *When the home receives all the invalidation acknowledgments and the data, it forwards the data to the processor in the NILocalPutX handler.*

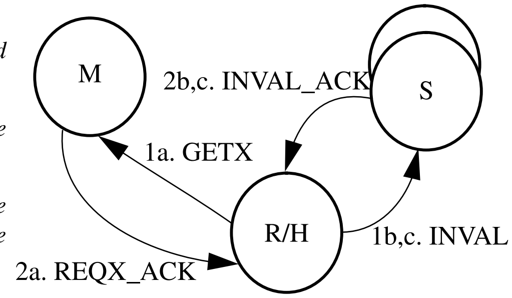


FIGURE 21a. Local write request satisfied by remote master. Replacement processing is omitted for simplicity.

- *The requestor (R) invokes the PIRemoteGetX handler, misses in the AM, and sends a GETX request (1) to the home (H)*
- *The home forwards the request (2a) to the master (M) and sends invalidations to each sharer (2b,c).*
- *The master sends the data to the home (3a).*
- *Each sharer sends an invalidation acknowledgment to the home (3b,c).*
- *When the home receives all the invalidation acknowledgments and the data, it forwards the data to the requestor (4).*
- *The requestor forwards the data to the processor in the NIRemotePutX handler.*

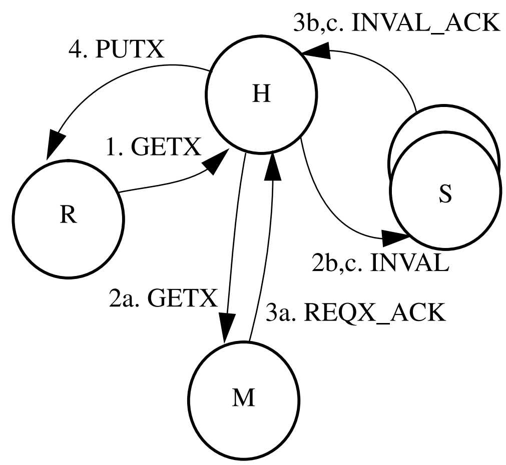


FIGURE 21b. Remote write request satisfied by remote master. Replacement processing is omitted for simplicity.

can return the line directly to the requestor using a special PUTX, so that the home knows it can send the data directly to the processor rather than storing it while waiting for invalidation acknowledgments to return.

Remote write request. First, the AM is checked in the PIRemoteGetX handler. If the datum is present, and is the only remaining copy in the system, it is immediately returned to the processor. Such a situation might arise if a processor requested a datum, wrote it, and then had the datum evicted from the cache to the AM, similar to RAC. If the datum is not present, the home is responsible for locating the master, procuring the data, storing it until all invalidations return, and then sending the data to the requestor. The requestor, in the NIRemotePutX handler, sinks the data and performs the necessary AM replacement processing. The scenario is depicted in Figure 21b. Again, if the master has the only copy of the datum, it can send the datum directly to the requestor, with an acknowledgment sent to the home so that the home can modify the master field in the directory, similar to DynPtr (as shown in Figure 14c).

Local Writeback. Local writebacks are more complicated for COMA than for RAC and DynPtr, since the home is not the static backing store. If a placeholder exists in the AM, then the datum is written back to the AM. If a placeholder does not exist in the AM, and if a free slot does not exist in the AM, a new master must be found for the line. Different nodes are queried in order to find a new master. Each prospective master is sent the datum–if it accepts the datum, an ack is sent to the home. If it cannot accept the datum, it NACKs the request and the home queries another node.

Remote Writeback. When a node needs to replace remote data, it invokes the PIRemotePutX handler. Unlike in DynPtr, PIRemotePutX first checks if there is a placeholder for the dirty datum in the AM. If so, the datum is written back to the appropriate location in the AM, and is not sent home. If there is no such placeholder, the home is sent a writeback request. The home must find a master, as in the local writeback case.

A number of details have been omitted for simplicity. For more details, the reader is referred to the complete specification for our COMA protocol[Sou98].

3.5.2 Characterization in the design space

Where is data replicated/migrated? Local and remote data are replicated both in the processor caches and in the AMs. Migration occurs in the sense that the backing store for lines can move to the node requesting the datum. Neither RAC nor DynPtr move the backing store.

Who decides when to replicate/migrate? Like DynPtr and RAC, the coherence protocol performs the replication/migration. The protocol replicates the data into memory/cache and migrates the backing store into memory. This replication and migration is eager: once the cache line is requested, it is immediately replicated/migrated to the requestor.

What happens when a conflict occurs? On processor cache conflicts, if the AM has a copy of the data, the line is sunk in the AM. Otherwise, a replacement request with the line is sent to the home, who is responsible for finding a master. Because the home may have to query a number of nodes before finding a new master, it requires pre-allocated space to ensure that it can resend such requests in the event of a NACK. If the home does not have enough room to pre-allocate space when it receives the original replacement request, it must NACK the replacement request. The requestor must therefore be able to regenerate replacement requests, and must pre-allocate space of its own. This issue is discussed in more detail when we discuss some of the challenges of COMA.

When a reply datum maps to the same location as a master copy (an AM conflict), the master line is sent to the home, who is responsible for finding a new master. Rather than blindly querying all nodes in search of a new master, we employ a heuristic first proposed in [JoH94]: the node supplying the reply datum presumably has an empty slot in its AM, corresponding to the slot that the reply datum used to occupy. This supplier node is queried first to see if it can accept mastership of the line. If it cannot accept the line, then others are checked. At this point, nodes currently sharing the evicted datum are given priority to become the new master. These nodes are not initially checked because it is a high overhead to consult the sharing list.

What is the resulting view of memory? Part of memory is used for backing store of PP protocol code/data, and for the AM tags/state. The rest of memory is split between reserved memory and memory allocatable by the OS, although all of memory is available for replication or migration. The amount of reserved memory can be statically partitioned, requiring minimal changes to the OS that mark pages as unallocatable and therefore reserved from use by the OS. Alternatively, the reserved memory can be partitioned dynamically, requiring more significant OS modifications. For example, whenever a page is allocated, the cache-coherence protocol would need to be notified in order to properly tag the cache lines on the page as master copies. A dynamic partitioning would be more adaptable to changes in memory usage across time, but the extra complexity might also hurt performance. For simplicity, in this thesis the reserved memory is statically partitioned.

What are the potential benefits over base cache coherence? Dynamically migrating/replicating all data can ensure that most capacity misses from the processor cache are satisfied without leaving the node. COMA does not rely on any initial static placement, unlike RAC, because locally-allocated data can move just as remotely-allocated data. There should be fewer handlers run at the requestor, and less contention at the home. The memory bandwidth requirements at requestor nodes will go up, since remote requests will access memory (unlike DynPtr) to see if the AM has a given datum, but memory bandwidth requirements at the home may go down because fewer requests reach the home.

What are the challenges in building this scheme vs. base cache coherence? While COMA offers improved locality through dynamic replication/migration of data, and also potentially reduces the number of redundant copies of data in the system, the improvements come at the cost of increased complexity. At the firmware level, COMA is by far the most complex of the protocols we study. As with RAC, there are changes in data structure organization. There are changes in all processor and network requests, not just remote ones. In addition, there are a vast number of other issues that must be dealt with in COMA, even on the simplest protocol transactions. We will discuss these changes in the next section.

3.5.3 Complexity Issues in COMA

By treating all of memory as a cache, COMA requires tags and state on each line in memory. This is one of many complexities of COMA versus CC-NUMA or RAC. On each processor cache miss in COMA, the tags and state must be checked at the requestor. At the home and at remote nodes, incoming requests must also check tags and state to locate a datum. In comparison to a simple CC-NUMA design, such checking adds overhead on each access, and complicates handlers, since on any node a line can be either in the cache or in the AM, and both may have to be checked.

HL	IO	P	T	WL	WN	H	L	Dev	RP	HP

FIGURE 22. HeadLink entry for COMA. The fields are explained in Table 6**.**

At the requestor end, remote read or write requests, which would otherwise be forwarded to the home automatically, must instead check the AM first on a processor cache miss. On an AM hit, we reduce runtime by saving the overhead of retrieving the data from another node, as would be necessary in a CC-NUMA design. However, on an AM miss the latency is increased relative to CC-NUMA because we perform the same actions that we would have done in DynPtr, with the addition of the AM checking overhead. At the home yet another tag check must occur, and again at a remote node if the data is not at the home. In contrast, in CC-NUMA, remote requests are automatically forwarded to the home, where some simple state (a local bit) must be checked to determine if the data is at home. If the data is not there, then the request is forwarded to a remote node, which, as noted above, must have the data in the cache, so that an intervention to the processor or a query to the cache can immediately be done.

To accommodate tags and state, and to track the location of the master copy of a datum, COMA needs different data structures from RAC and DynPtr. While both RAC and COMA have a notion of memory as a cache, their needs are fundamentally different. The primary data structure differences between COMA and RAC are that COMA needs to track a master copy and also must maintain tags and state on all lines in memory, not just remote lines. As in RAC and DynPtr, COMA uses a singly-linked list to keep track of sharers. One sharer is kept in the headLink, and the rest of the sharers are kept in a separate sharing list. The sharing list is organized in the same way as in DynPtr and RAC. The directory information, however, is changed with respect to RAC and DynPtr, in order to accommodate master tracking and other important issues. The headLink entry for COMA is shown in Figure 22, with the fields briefly described in Table 6. The important differences between the headLink entries of DynPtr and COMA are the following:

- *Transient* bit: This bit is set if a replacement has happened while a protocol transaction is pending. For example, if node P1 requests a line, and while the home is seeking that line, the home receives a replacement request for the line from node P0, the transient bit is set. This bit helps the home determine when a transient protocol condition has arisen, so that the home can change the master field in the headLink appropriately. These transient issues will be discussed in more detail in Section 3.5.3.3.
- *WriteLastReq* bit: This bit is set if the most recent transaction from a remote node was a write. It is used to prevent a remote master from surrendering data incorrectly. In particular, if a master is expecting write ownership of a line, and receives a request for that line before the ownership arrives, then the master should not surrender data. This bit is used at the home to send the appropriate GET message to the master. This GET makes sure to determine if the master is expecting ownership before grabbing the data from the master.

Field	Full Name	Field Width (bits)	Description
HL	HeadLink	19	Pointer to first element in linked list of sharers
IO	IO	1	Set if the only valid copy is in the I/O system of some node
P	Pending	1	Set if the line is in the midst of a protocol operation
T	Transient	1	Set if a replacement has happened while a protocol transaction is pending
WL	WriteLastReq	1	Set if the last request to a line was a write
WN	WriteNak	1	Set if a write has been NACKed but invalidations are outstanding
H	HomeOnList	1	Set if the home is caching the line or has it in its AM
L	List	1	Set if there is a sharing list for the cache line (in addition to Hptr)
Dev	Device	2	Contains the device number of the IO device requesting this line
RP	RealPtrs	11	Contains the number of sharers in the sharing list
HP	HPtr	11	Identity of the master node

TABLE 6. Description of the fields of the headLink entry for COMA.

- *WriteNak* bit: This bit is set if a write request has been NACKed while invalidations are still pending. This bit helps the home determine if it is responsible for sending data when all invalidation acknowledgments have returned.
- *HomeOnList* bit: This bit is set if the home is on the sharing list. The home may have the data in its cache, its AM, or both.
- *Hptr* field: This field stores the identity of the master. Hptr is short for HeadPtr, because the master is considered the first (or head) pointer in the sharing list. In contrast to the other protocols, in which there is no distinction between the first sharer and the other sharers, in COMA the first sharer is treated specially for tracking purposes.

While each line has 64 bits available for the headLink, the directory information does not take up the full 64 bits. Because COMA requires tags and state to be stored for each memory line, and because a headLink is also required for every line in memory, we use the unused bits in the headLink to store the tag and state information for the AM. Conceptually, each node contains headLink information for all the lines that it is home to, and also contains separate tag and state information for each line in the AM. We use an indexing based on the AM address to find the appropriate AM information. By storing the tag information in the unused bits of the headLink, we save having to allocate extra storage above and beyond the headLink.

The tag and state information required for COMA is the same as for RAC, except that there is no need for an Offset field: since the AM is direct-mapped, a given line can reside in only one location in another node, and no two lines from the same node can map to the same location on another node. Table 7 shows the tag/state

Field	Full Name	Field Width (bits)	Description
AmP	AmPending	1	Set if any operations are pending on this AM line
I	Invalidate	1	Set if an invalidation has been received for this AM line while a request is outstanding for this line
AmT	Amtag	8	Stores tag information
AmS	Amstat	3	Stores state information

TABLE 7. COMA tag and state information.

AM state	Description
INVL	AM line is empty
SHAR	AM has shared non-master copy of line. Cache has no copy.
SHARS	AM/cache both have shared, non-master copy of line.
SHARM	AM is master, and line is shared. Cache does not have a copy of the line.
SHARMS	AM is master, and line is shared. Cache also has a copy of the line.
EXCL	AM is master and has the only copy of the line. Cache has no copy.
EXCLS	AM and cache have copies of the line, and AM is master. This node has the only copies of the line.
EXCLX	Data is dirty in the cache, and AM has a placeholder for the line.

TABLE 8. Description of AM states.

information required for COMA. There are additional states required for COMA because of the presence of the master copy. For example, a master node may have a datum shared, and the home may not be a sharer. This situation cannot exist in DynPtr or RAC. The states required and their meanings are shown in Table 8.

The tag and state checking is just one facet of COMA complexity, and is not the dominant component. COMA is more complex than other schemes primarily because it lacks a static backing store. The lack of a static backing store complicates the replacement process, which affects a number of basic protocol transactions. Unlike DynPtr and RAC, in which a writeback from a processor is unquestionably sunk at the home, in COMA the writeback reaching the home is only the beginning of the replacement process. To prevent losing the backing store copy of a line, one copy of the line must have been designated the master copy, and is carefully tracked. When the master is replaced, and the replacement reaches the home, the home is now responsible for finding a new master, either by becoming the master itself or by querying other nodes to find a new master.

- *The requestor (R) sends off a read request (1) for line A to the home of A (H_A).*
- *The home of A forwards the request (2) to the master of A (M_A).*
- *The master sends the data reply (3a) to the requestor and an acknowledgment (3b) to the home.*
- *Line A conflicts with the line currently in the AM at the requestor, line B. When the PUT A reply (3a) arrives, the requestor must generate a replacement request for line B (4) to the home of B (H_B). The PUT reply for A therefore requires a request to be generated, and we thus have a reply/request chain. A reply/request chain has potential for deadlock.*
- *The acknowledgment to the replacement (5a), and the subsequent search for another master are needed to complete the replacement process (5b,6). If the replacement request (5b) is NACKed by the prospective master, the replacement process must be restarted, creating more reply/request deadlock scenarios.*

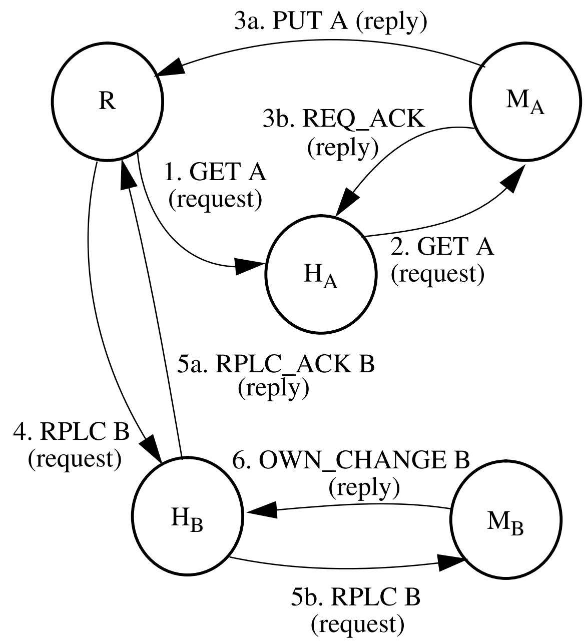


FIGURE 23. Potential deadlock related to replacement processing in COMA.

The process of finding a new master introduces several complexities into COMA. The first set of changes are described above, and involve both creating data structures for tracking the location of a master and the state of a given line, and protocol-level changes to perform the tracking and state manipulation. In addition to data structure reorganization and tag checking, there are also issues with resource allocation, unsolicited data receipt, and transients induced by requests occurring during replacement events. We now discuss each of these issues in turn.

3.5.3.1 Resource Allocation

In COMA, when a node requests a datum and receives a reply, the reply may displace a master copy in the AM. This master copy must be sent to the home so that the home can find a new master, as shown in Figure 23. For the home to find a new master, the home must be able to generate a network request. If the home is somehow unable to guarantee that it can generate a network request, it must NACK the replacement request from the old master. The old master must in turn be able to regenerate its replacement request.

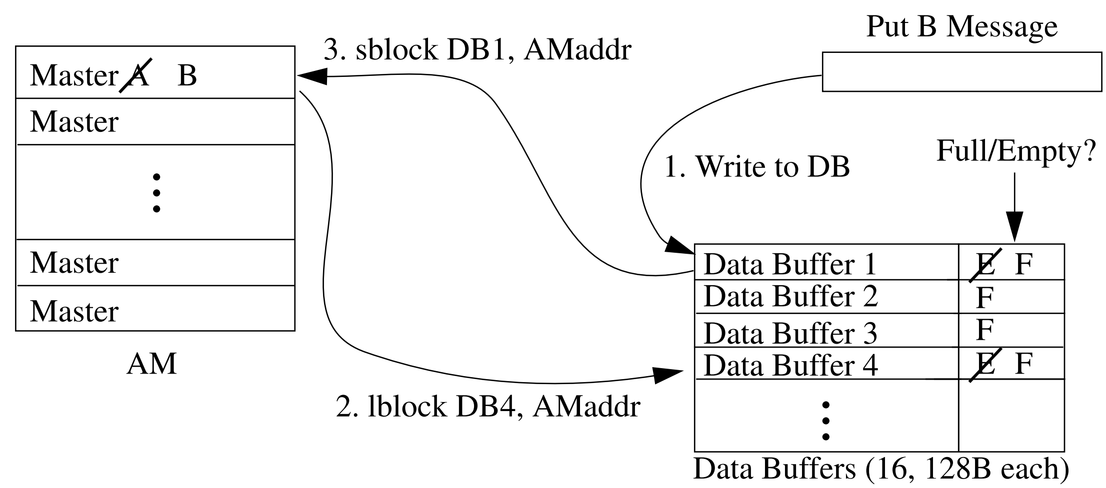
Consider the resources required in the preceding scenario. The requestor must generate its original replacement request (message 4 in Figure 23) in response to a reply message (message 3a in Figure 23). This is a reply/request chain in the protocol, which contains the potential for deadlock if resources are not allocated properly. For example, suppose the reply could not be sunk because the replacement request could not be generated due to insufficient outgoing request queue space. If every node were in this situation, then we have deadlock, because no incoming replies can be sunk to guarantee forward progress in the protocol. Another possible deadlock involves the node receiving the requestor's replacement message. This node (node H_B in Figure 23) must perform two actions: first, it must accept the requestor's replacement message (message 4) and generate a reply to that message (message 5a); second, it must generate a request (message 5b) to the prospective master. A request (message 5b) is generated in response to a request (message 4), which is another possible deadlock. For example, suppose no node can receive an incoming request because it cannot generate an outgoing request. Then each node would be unable to make forward progress. Even if deadlock were not the concern, if the replacement requests (messages 4 and 5b in Figure 23) could not be generated for any reason, then the datum must be stored somewhere or it would be lost.

To guarantee that the requestor can generate a replacement request, the requestor must thus preallocate space in the continuation queue (i.e. the SWQ) before even sending its original data request. If it is unable to guarantee SWQ space, it must NACK the processor's data request, and the processor must retry the request until MAGIC has SWQ space. Once this SWQ space is allocated, the requestor is guaranteed that it will eventually be able to generate a replacement request on this line.

There are two resource-related reasons that a replacement might not be able to be generated. The first, as mentioned above, is that there is insufficient outgoing request queue space to generate the request. The second reason has to do with the data storage resources in FLASH. NI handlers are guaranteed a data buffer to store any incoming data associated with the network message. However, if a replacement request must be generated, then the data to be replaced must be read out of memory and stored in another data buffer so that it can be sent off-node. If there is no free data buffer for storing the replacement data, then the replacement cannot be sent. Instead, the replacement data must be written word-by-word to a SWQ storage area using word-sized loads and stores to memory (Figure 24b) rather than cache-line-sized loads and stores to MAGIC data buffers (Figure 24a).

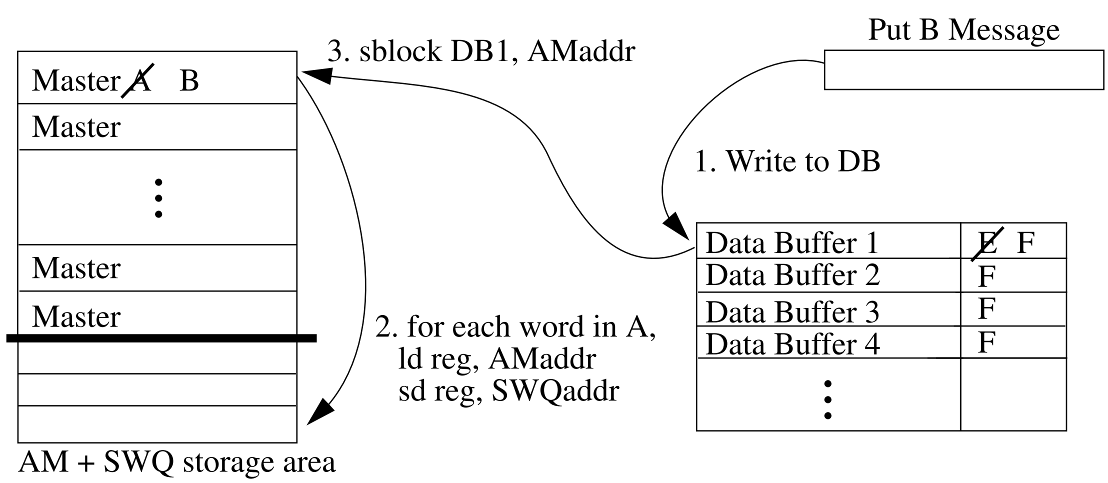
Once the requestor guarantees that it can resend any NACKed replacement, it may seem that there are no further resource problems, because if necessary, the requestor could be put in charge of the replacement process, rather than the home. However, even if the requestor eventually finds a new master, it must notify the home as to the new master's identity so that the home can track the location of the master. Moreover, if the requestor were in charge, the home would have to somehow send the sharing list to the requestor so that the requestor could try sharers from the list to be the new master. Sending a sharing list would be an inefficient operation in FLASH, requiring a list traversal at the home and then a list addition sequence at the requestor. As a result, it is more efficient to perform replacement processing at the home, so we choose to let the home handle replacements and deal with the corresponding resource issues.

Because the home is in charge of a portion of the replacement sequence, and because its replacement requests may get NACKed by prospective masters, the home sets aside SWQ space when it receives the



- *The data in the incoming message (line B) is written (1) to an empty data buffer. The data buffer is marked full.*
- *Line A, the data at the AM offset that B is to occupy, needs to be evicted. Line A is copied (2) to an empty data buffer using a single nonblocking PP lblock instruction.*
- *Line B is copied from its data buffer to the AM using a single nonblocking PP sblock instruction (3).*

FIGURE 24a. Replacement processing, data buffer available for current master data.



- *The data in the incoming message (line B) is written (1) to an empty data buffer. The data buffer is marked full.*
- *Line A needs to be evicted. There are no free data buffers left, so line A is copied (2) word-by-word to the software queue, using blocking loads and stores to and from PP registers.*
- *Line B is copied from its data buffer to the AM using a single nonblocking PP sblock instruction (3).*

FIGURE 24b. Replacement processing, data buffer unavailable for current master data.

original replacement from the requestor so that it can continually resend replacement requests, if necessary. If the SWQ space cannot be set aside, the replacement request is NACKed back to the requestor. If it can be set aside, the requestor is sent an acknowledgment so that it can deallocate its on SWQ space. If a replacement request sent off by the home is NACKed and no outgoing request queue space exists, then the replacement is queued on the SWQ by the home.

Notice that the SWQ space is used only if data buffers or outgoing storage space is not available. The data buffer and outgoing queue checks are expensive to perform in MAGIC, leading to increased occupancies on many replies. However, these checks are less expensive than enqueuing and dequeuing off the SWQ. Repeated NACKs of the replacement request from the home also consume PP occupancy, since a new replacement request must be generated each time, requiring outgoing queue and buffer checks. These resource checks consume a large portion of code overhead compared to DynPtr, since each request handler must allocate space, and each reply must check space limitations. The runtime PP overhead is seen in several places. First, PI requests must check SWQ space before even forwarding requests to the home, resulting in higher occupancy over and above the simple tag and state checks required. The overheads are also seen when considering Put handlers for COMA vs. DynPtr. For DynPtr, a Put handler simply sends the data to the processor. In COMA, the data is sent to the processor, and then the AM must be checked to see if replacement processing is required. If replacement processing is required, the resource checks must occur and the data currently in the AM must be sent to the home. Only after this replacement is sent to the home can the incoming data be placed in the AM.

3.5.3.2 Unsolicited data receipt

Another major complexity with finding a new master involves unsolicited data receipt. The previous discussion considered replacements from the point of view of the home and the original requestor. Now let us consider what happens at a potential master. A potential master AM receives an unsolicited request for a datum. This potential master will only accept the datum if it can do so without displacing another master copy. Otherwise a ping-ponging of master copies would result, leading to a great deal of network traffic per replacement. If the datum is accepted, then an acknowledgment is sent to the home so that the home can deallocate its SWQ space.

Such unsolicited data receipt can cause some interesting problems that would not occur in a typical CC-NUMA design. One problem is related to the inclusion property used by the AM. If we do not enforce inclusion between the AM and the cache, then it is possible for a node to receive a master transferral request for a datum that the node has in its cache (but not in its AM). Because the datum is not in the AM, the AM tags and state do not have a record of the line. Either the PP has to send an intervention to the processor to determine if the line is in the processor cache (a very costly operation), or it must make the conservative assumption that the line is in the cache and designate the AM state to indicate “shared master, data also in cache.”

Unsolicited data receipt also introduces problems with generating replacement messages. When a node requests a datum, it waits for the reply to return before generating any replacement/writeback that may be necessary. It could generate the replacement immediately after generating the outgoing data request, but if it receives another replacement in the meantime, then when the requested datum returns, it would again have to send off a replacement. Thus, to avoid sending multiple replacements, our policy is to perform replacement processing only when the requested datum returns. Note that this policy is different from the processor cache's policy of generating replacements. The R10000 processor cache is fill-before-spill[MIP95]: when a datum is requested, the data request is sent to MAGIC, and immediately thereafter the replacement request is sent to MAGIC. This replacement can be generated immediately because the processor cache never receives unsolicited data, and thus the data reply will never have to evict any line from the cache before being sunk. While one way to avoid sending multiple replacements might be to generate a replacement immediately after a read/write request goes out, and then reject any incoming replacements until the reply returns, there is a problem when the data coming via replacement is the actual datum requested. Rejecting this datum might mean that the requesting node never receives its requested datum, a possible livelock.

One final problem that we will discuss concerning unsolicited data receipt relates to finding replacement data that is in transit. A node may request a line, but if that line is being shipped to other nodes in an attempt to find a master, then the data request will go to the home, who will forward the request to the supposed master. The supposed master may NACK the request because it does yet not have the data or could not sink the mastership request. The data request will be NACKed back to the requestor, who must regenerate the request. This situation can happen repeatedly while a new master is being found, and the home must be careful in tracking the master so that the requests are forwarded properly while the data is in transit. If a replacement request is forwarded back to the node requesting the replaced line, there are two possible outcomes:

- The requesting node may accept the data via replacement before its data request reaches the home. The home will then forward the request back to the requestor. This protocol condition would be an error in most CC-NUMA protocols, because a node cannot receive unsolicited data, and therefore cannot unexpectedly receive a line that it has just missed on. In COMA, to accommodate this condition, NIRemoteGet/GetX/Upgrade handlers must check the source of a request, and send the reply on the appropriate outgoing queue (i.e. to the processor or to another node) accordingly, updating AM state if the source of the request is the same as the destination.
- The requesting node may reject the replacement data due to lack of free AM space. Because the AM keeps no state concerning outstanding requests, it is possible that the requestor may NACK the mastership request for the very data it desires simply because it already has a different master at that location. To attempt to sink the line would require some method of evicting the master line currently in the AM and sending a replacement message to the home. All of the problems with a requestor sending a replacement would then be possible.

- *The requestor (R) sends off a read request (1) for line A to the home of A (H).*
- *The home of A forwards the request (2) to the master of A (M).*
- *The master sends the data reply (3a) to the requestor and an acknowledgment (3b) to the home.*
- *The acknowledgment (3b) is delayed in the network. The requestor receives the PUT reply (3a) and replaces line A before the home receives the acknowledgment.*
- *The replacement (4) from the requestor arrives at the home before the acknowledgment (3b). The home must find a new master and make sure to ignore the acknowledgment to keep the master information correct.*

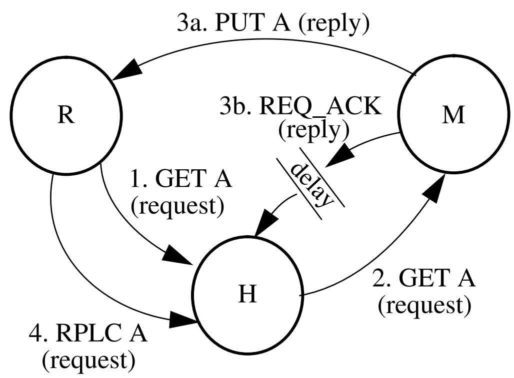


FIGURE 25. Master replacement before home is notified of new master.

3.5.3.3 Transients and Other Challenges

Master replacement before home notification of new master. The unpredictability (with respect to outstanding data requests) with which replacements can be received introduces another transient condition worth mentioning. This situation results from the varying latency of different messages in the network. Consider a simple read request that is satisfied by the master. The master sends the data reply to the requestor, as well an acknowledgment to the home that the master has changed. When the acknowledgment is received, the master field at the home is updated. A similar situation occurs in CC-NUMA: when a remote node satisfies a write request, it sends an acknowledgment to the home indicating that the requestor now has the line dirty. When the home receives the acknowledgment, it modifies its state tracking the dirty outstanding copy.

The complexity that occurs in COMA is that the requestor may receive and replace the line before the home receives acknowledgment that the requestor is the new master. The replaced line is sent to the home, who may receive it before the mastership transferral acknowledgment from the previous master (see Figure 25). The home must maintain some state so that it can determine when this situation has occurred, and ignore the acknowledgment. While waiting for the acknowledgment the home must also find a new master node. Mastership can be transferred on reads, writes and upgrades, so the home must maintain state on each of these transaction types to avoid problems with master tracking. In contrast, in CC-NUMA, ownership is transferred only on write requests, so there are fewer such cases to consider. Moreover, in a CC-NUMA design no transient bit is needed because the replacement from the requestor is immediately sunk at the home, forcing the line to be shared. When the acknowledgment is received, the home checks if the line is dirty or shared. If the line is shared, the home knows that the replacement has occurred first and ignores the acknowledgment.

Writes. The previous sections have focussed on the difficulties in maintaining master copies in the presence of replacement traffic. One additional obstacle that a COMA implementation must address is related to maintaining master copies across write transactions. In a read transaction, the home can satisfy a request even if it is not master simply by not transferring mastership status on the requestor. For writes, however, the mastership must be transferred since the requestor will become the last copy of a line. In order to prevent the complexities of storing two masters temporarily and disambiguating between the old and new master, we require that on a write miss, the datum is retrieved from the current master.

Forcing write misses to be satisfied by the master creates some interesting protocol conditions that are similar to those encountered when processing replacements. One prominent example is that when a master node incurs a write miss on a datum, but does not have an exclusive copy of the line in its AM (i.e. it is a shared master), the request must still be forwarded to the home. The home locks the directory and must then forward the request back to the master, who happens to also be the original requestor. Here again we see a request forwarded back to the original requestor. The original requestor/master then performs the necessary processing.

One additional difficulty with writes is resource-related. In a protocol that is used in conjunction with a sequentially-consistent memory model [Gha95], write data can only be returned to the processor when all outstanding invalidation acknowledgments have been received by the requestor. In a CC-NUMA design, when a datum is shared, the home has a copy. Thus, if the home receives a write request and the datum is clean, it can send out all invalidations, collect the acknowledgments, and then send the data to the requestor. If the datum is dirty, then either the home has it dirty or a remote node has it dirty. Either way, there are no invalidations required, and either the home or the remote node forwards the data directly to the requestor.

In COMA, the situation changes because of the lack of static home and because of the master copy. A master copy may be a shared line. Thus, on a write miss, a request may be forwarded to a remote master node even if the line is shared. In contrast to CC-NUMA, in which forwarding to a remote node implies that the data is dirty, in COMA, the datum may be clean. There are two options. Either the master can forward the data to the requestor, and the requestor must store the line until it receives all invalidation acknowledgments, or the master can forward the data to the home, and the home stores the data while awaiting for the acknowledgments, eventually forwarding the data and acknowledgment in one message to the requestor. We choose to let the home store the data because all of the directory information is already at the home, so keeping track of outstanding invalidations requires no additional state beyond that in the directory. In either scheme, however, data may have to be stored somewhere until all invalidation acknowledgments have been received. This situation is different from CC-NUMA, because CC-NUMA has a static backing store for data. In CC-NUMA, if a line is clean, then the home has a copy. Once all invalidation acknowledgments have been collected, the home can read the clean data out of memory and send it to the requestor. In COMA, the home has no such static backing store. Instead, the home must explicitly allocate space for the data, collect both the line from the master and invalidation acknowledgments from other requestors, and then send the data to the requestor. If the home cannot allocate space for the data, then it cannot service the write request. The home uses the SWQ to store data: before the home forwards a write request to a master, the home attempts to allocate SWQ space. If this SWQ space cannot be guaranteed, the original write request is NACKed. Even if SWQ space is available, enqueuing and dequeuing data from the SWQ can be quite expensive, requiring numerous word-sized loads and stores, so writes can be somewhat expensive.

3.5.3.4 Static Protocol Complexity

Table 9 lists representative latencies for our COMA implementation. Unlike DynPtr and RAC, these numbers can vary quite widely even in the absence of contention, since in COMA, a datum that is in the AM can be in any number of states. For example, a datum can be a shared master (i.e. it is master, but there are also other sharers of the line), or it can be exclusive master (i.e. it is the only copy of the line), or it can be simply shared. In addition, in all of these preceding cases, it can be in the AM alone, or in both the AM and cache. The latencies/occupancies in the table are for representative cases, and can be less than or greater than the actual latencies/occupancies. In RAC there is less variance because there is no notion of a master copy, so there are fewer different states that a line can be in.

Table 9 provides a quantitative measure of COMA complexity compared to DynPtr and RAC. Many of the COMA handlers are more expensive than their counterparts in RAC and DynPtr. Consider the difference in occupancy between NILocalPutX for DynPtr/RAC and NILocalPutX for COMA. In DynPtr and RAC, an NILocalPutX costs 11 cycles of occupancy. All this is required is to forward the data to the cache, which costs 1 cycle, and then update the directory, which costs 10 cycles. Updating the directory involves clearing the pending bit so other operations can now occur on the line, setting the dirty bit so it is known this line is dirty, setting a local bit so the home knows that it is a sharer, and clearing an IO bit since the data is not in the IO system. Because local lines do not go in the RAC, DynPtr and RAC have identical latencies and occupancies for this handler.

In COMA, the situation is very different. The line is first forwarded to the cache. Next the line is written to the AM. However, before the line can be written to the AM, whatever is currently in the AM (if anything) must be evicted. After the eviction, the home must update the directory by clearing the pending and IO bits, setting a local bit indicating that it has a copy of the line, setting the tag in the AM, and also setting the identity of the master to be itself.

To understand quantitatively the complexity in COMA it is important to consider the eviction process, since eviction is the primary contributor to occupancy in Put/PutX handlers for COMA. As mentioned earlier in Section 3.5.3.2, eviction must occur when a requested line is received, rather than when the request initially goes out. The eviction process involves several stages. First, the PP must make sure that there is outgoing request queue space in order to generate the replacement. The eviction is a request, and it is generated as a

Handler	Operation	Latency	Handler Occupancy
PILocalGet	request satisfied locally (AM hit)	11 (+2)	13 (+2)
	forward request to remote owner (AM miss)	23 (+10)	32 (+18)
PIRemoteGet	request satisfied from AM	10 (n/a)	12 (n/a)
	forward request to home (AM miss)	16 (+13)	17 (+13)
NILocalGet	forward data in AM to requestor	15 (+5)	21 (+7)
	forward data in cache to requestor (does not include intervention time)	34 (+2)	39 (+6)
	forward request to remote owner	15	18 (+2)
NIRemoteGet	retrieve data from cache and send data to requestor (intervention not included)	28 (+18)	37 (+23)
	retrieve data from AM and send data to requestor	18 (n/a)	28 (n/a)
NILocalPut	send data to processor (latency), update directory including sharing list (occupancy)	1	88 (+77)
NIRemotePut	forward data to processor and AM	1	56 (+54)
PILocalGetX	request satisfied locally (AM hit, no sharers)	16 (+5)	18 (+5)
	forward request to remote owner (no sharers)	19 (+5)	31 (+16)
PIRemoteGetX	request satisfied from AM	15 (+12)	17 (+13)
	forward request to home (AM miss)	18 (n/a)	23 (n/a)
NILocalGetX	forward data in memory to requestor (no sharers)	20 (+8)	31 (+11)
	forward data in cache to requestor	40 (+10)	53 (+19)
	forward request to remote owner (AM miss)	20 (+5)	26 (+7)
NIRemoteGetX	retrieve data from cache, send data to requestor, and send ack to home (intervention not included)	33 (+16)	40 (+17)
	retrieve data from AM and send data to requestor	22 (n/a)	34 (n/a)
NILocalPutX	send data to processor (latency), update directory (occupancy)	1	50 (+39)
NIRemotePutX	forward data to processor and AM	1	48 (+46)

TABLE 9. Representative latencies and occupancies of common protocol handlers in COMA. The differences, if any, between RAC and DynPtr latency/occupancy values are shown in parentheses next to the corresponding latency/occupancy. n/a means not applicable. Local accesses to the AM are considered analogous to local memory accesses in DynPtr. .

result of a reply (the data response). MAGIC cannot wait for outgoing request space to be free before accepting the reply. Such waiting could lead to deadlock: suppose none if the nodes had any outgoing request queue space, and each node had a data reply incoming. None of the nodes would be able to accept their reply, and thus none would make forward progress. This is an example of the reply/request cycle alluded to in Section 2.5.2. Instead, MAGIC must accept and schedule this data reply handler despite the status of the outgoing request queues, and let the PP code check for queue space. If that queue space is unavailable, the eviction must be placed on the SWQ and be performed later. Clearly, even before a request is forwarded from a requesting node to either the home or the current master, the requestor must be able to allocate SWQ space so that any eviction that may need to be generated will have SWQ space available. If no such space is available, the original request cannot be sent off-node by the requestor. Instead, the request must be NACKed to the processor and eventually regenerated by the processor.

After the check for outgoing queue space, the PP must then check for a free data buffer in which to place the AM data. The incoming data that is about to occupy this slot in the AM is waiting in its own data buffer, and cannot be placed in the AM until the current data is evicted. If there is a free data buffer available, the current AM data is read into it, and the replacement request is generated. If there is no free data buffer available, then the AM data must be written to the SWQ and the replacement is generated later, as shown in Figure 24b.

Checking for queue space takes about 6 cycles: the loading of the appropriate state takes 3 cycles and the test takes 3 cycles (the branch is two cycles and setting up the test is a single cycle). Similarly, testing for free buffer space take about 5 cycles. Once queue space and buffer space are established, the replacement message must be prepared, which can take as much as 7 cycles: one cycle each to set up source and destination, one cycle to issue the load from AM to data buffer, one cycle to set up the buffer number that will store the data, another cycle to insert the buffer number into the message header, one cycle to set up the message type, and finally, one cycle to perform the send itself. So far, we have counted an eviction as taking 18 cycles, and this does not even count the tag check and state check that determine whether an eviction is required. The tag check itself requires about 7 cycles to prepare the tag address and load the tag, and then 2 cycles to perform the tag check. The tags and state are kept in the same data structure, so the state check requires a single cycle to set up which state to check for, and another 2 cycles to perform the branch (a branch has a load delay slot). So the total so far is 30 cycles for eviction for the specific AM case that we have selected: if the data were in another state, then the total would be different. Other instructions, like any sort of stack manipulations that may be required prior to function calls, increase the total by a few cycles to around 33 cycles.

Once the eviction is done, the data must then be written to the AM, and the AM tag and state must be modified. Also, because this handler runs at the home, the directory state must also be updated. Updating the AM tag takes 2 cycles (involving a shift and insertion of bits from one register into the appropriate place in the AM tag), and updating the state takes 2 cycles. Updating the directory to indicate that the home is now master, that there are no other sharers, and that all operations on this line are complete, takes 7 cycles. There are a few other cycles again related to stack manipulations or decrementing SWQ space, that add 7 more cycles.

Another important high-occupancy handler is NILocalPut, which consumes 38 more cycles than NILocalPutX. 36 of these 38 cycles are required for adding sharers to the sharing list. Here again we see the complexity of a COMA design causing increased occupancy. In each protocol we have considered, the headlink stores the first sharer in the sharing list, while a sharing list is used for subsequent sharers. In DynPtr and RAC, there is no difference between the first sharer and subsequent sharers. In COMA, however, one sharer must be designated the master. Since the master field is accessed relatively frequently, it is stored in the headlink, similar to the first sharer in the other protocols. When a node becomes the master, it must be stored in the master field. First, however, the current master must be added to the sharing list. This sharing list addition is required in NILocalPut because the home is becoming the master. In other protocols, however, no special sharing list manipulation is needed in NILocalPut, since each sharer is the same as any other sharer, and the home is designated as a sharer simply by setting a local bit. One might choose simply to have a HomeIsMaster bit in COMA and avoid adding the previous master to the sharing list, but this would require code to differentiate between the master field storing the actual master and it storing simply another sharer. Moreover, this would only help when the home becomes the master, which may not be the common case.

As seen by delineating the operations required on NILocalPut/NILocalPutX, the flexibility of MAGIC allows arbitrarily complex protocol manipulations, although there can be a high cost to such operations. The resource allocation checks and actions required on resource failure take up a large percentage of both static and dynamic code size. Unlike RAC and DynPtr, which each have been 8000-9000 lines of code, COMA has approximately 15000 lines of code. While this is purely a subjective figure of complexity, and the code size could be decreased by more efficient function calls or by more source- and assembly-level optimizations, the fact remains that COMA is more complex because of the lack of static home and the measures required to track data. In terms of handler count, COMA contains 87 handlers, all of which differ from DynPtr and RAC. 13 new NI handlers are required to deal with servicing replacements. There are 7 more SWQ handlers in COMA because the SWQ must be used for more resource conditions than DynPtr and RAC. The SWQ handlers in DynPtr and RAC are used primarily for resource conditions in which the PP is unable to send all invalidations at once: if the PP has insufficient outgoing request queue space to send an invalidation, it enqueues the remaining pending invalidations onto the SWQ to be sent later. For COMA, handlers are required not only for sending invalidations, but also for sending replacements if the PP has insufficient outgoing queue space, for sending replacements that could not be sent due to insufficient buffer space for replacement data, and for resending replacements that were NAKed for any reason. In total, the 12 SWQ handlers comprise about 1600 lines of code, compared to 5 handlers comprising about 500 lines of code for DynPtr and RAC.

Given the complexity issues with regard to COMA, the issue of MAGIC ISA suitability is more difficult to assess for COMA than for RAC. Clearly the latencies are higher, and in some cases much higher, than those for DynPtr or RAC. A number of these increased latencies could be reduced, although not eliminated, by specific MAGIC structures. For example, as in RAC a hardware lookup table can decrease tag and state check overhead. In addition, the ability to perform actions like resource acquisition in parallel with such state and tag checks would improve performance. But COMA genuinely has more bookkeeping to perform than other protocols, and ISA extensions can only help to a limited extent. When we examine the performance results we will see how improved locality succeeds at overcoming the complexities of COMA.

3.6 OS-based page migration/replication

The final scheme we consider is OS-based page migration/replication. We use the implementation done by Verghese *et al* in [VDG+96]. We call this scheme MigRep. MigRep differs from the previous three approaches by using system software, rather than the coherence protocol, to dictate the migration/replication policy decisions. Because the OS allocates and deallocates at the granularity of pages, migration and replication occurs in page-sized chunks. Using the OS allows policies potentially to be more complex than those implemented in hardware, since the OS can easily store history information to guide its decisions. The MigRep scheme is implemented in the kernel of a commercial OS, IRIX5.3. A complete discussion of this protocol is beyond the scope of this thesis, and the reader is referred to [VDG+96] and [Ver98] for more details.

The underlying coherence protocol is identical to DynPtr, except for a small overhead to count per-page-per-node cache miss rates. To reduce run-time overhead, we do not run the full instrumented handler on each handler invocation. Instead, we use *sampling*. With sampling, a counter is associated with the instrumented handler. When this counter reaches, say, 10, the full instrumented handler is run; in all other cases, only the normal operations are performed. The counter increment is placed in unused PP instruction slots so that it adds no overhead to the handler. By using sampling, only one out of every 10 handlers runs the instrumented handler to update the miss statistics. Sampling has been shown to be effective at lowering overheads without compromising accuracy [VDG+96]. The rest of the migration and replication logic is executed in software by the OS kernel.

The virtual memory subsystem (VM) of the OS is modified to support the migration and replication of pages. Because it is implemented in software, there is a fairly sophisticated policy that decides when to migrate or replicate a page. This decision is driven by a set of cache miss counters (stored in memory) that monitor the accesses to each page from the processors and generate an interrupt to the OS when a “hot” page is detected. MigRep therefore is responding to longer-term locality trends, unlike the other three protocols, which immediately replicate or migrate upon use. Currently, MigRep does not migrate or replicate kernel pages because the IRIX5.3 kernel on which it is based is not mapped through the TLB. The main sources of kernel overhead in MigRep are: fielding the interrupt from the cache-miss counting code, making the policy decision for the page, allocating and copying the page, and maintaining page coherence through locks and the flushing of TLBs.

MigRep replicates or migrates data to local memory at page granularity in response to excessive cache misses to a page. Excessive remote read misses to a page by numerous processors, without any intervening writes, characterize the page as a candidate for replication. Excessive remote misses (read or write) by a single processor relative to other processors characterizes a page as a candidate for migration. When the protocol decides to move or copy a page, the home of the page and the data become local to a node. In the case of replication, the VM creates a local virtual-to-physical mapping (read-only) and copies the datum into the memory of the requesting node. Subsequent requests are thus satisfied by local memory rather than remote memory. There are now effectively multiple backing stores for the same virtual address: one on each node that contains a replica. On a write, a TLB write access fault occurs and the OS invalidates other pages so that the requestor has exclusive access to the appropriate lines and preserves the single-writer multiple-reader semantics. In page migration, the VM again creates a local virtual-to-physical mapping, and also invalidates all other copies of the page. Because the backing store is then officially moved to a new location by changing the location of the home and directory information, all TLBs must be flushed to prevent them from using the old mapping.

MigRep differs from RAC and COMA, which just cache the data in local memory or move the physical backing store, but do not move the directory information (i.e. the home). In MigRep, the data is remapped locally and the directory is moved, so future accesses to the migrated or replicated line are actually local accesses, and the handlers that run are PILocal handlers[4]. In RAC and COMA, however, the backing store is local, but the home is remote, so the handlers that are invoked are PIRemote handlers. MigRep thus migrates/replicates the backing store as well as the directory information, and uses OS software to help enforce coherence. The hardware is effectively unaware of the presence of the MigRep code. All the protocol sees are local transactions that were formerly remote transactions. For page replication, the hardware is unaware of the existence of other replicas because the OS handles replica reclamation, so the underlying coherence protocol need not be changed to perform any replica page tracking.

MigRep treats all of main memory equally. Unlike RAC, memory need not be statically partitioned into memory used as a cache and memory used as backing store: all of memory can be used in either way. Moreover, there are no memory regions reserved from OS allocation, as in COMA. It is important, however, to respond to memory pressure and not cause paging because of excessive replication. This is done at the policy level by reducing replication in the face of memory pressure, and through selective reclamation of replica pages. Thus the MigRep scheme attempts to be robust to memory pressure.

3.6.1 Sample Protocol Transactions

Let us now consider some sample transactions for MigRep:

---

4. Actually, the directory itself is not moved. Instead, a new directory on the receiver node is used, and the directory on the sending node can subsequently be reused for another address.

Local read request. This case is identical to DynPtr.

Remote read request. This case is identical to DynPtr, unless the remote miss causes the miss counter at the home of the line to cross the replication threshold. If this occurs, the requestor OS is interrupted, and decides whether or not to replicate/migrate the page. The appropriate cache lines are copied to local memory[5] and a local virtual-to-physical mapping is made by the OS. Future read references are now PILocalGet.

Local write request. If the local page is a replica, then the TLB has the page in a read-only state, so a write fault occurs. The OS invalidates other replicas, flushes all TLBs, and allows read/write access to the page for the requesting node. If the local page is not a replica, then the actions are the same as for DynPtr.

Remote write request. If this miss causes a migration event, then the appropriate page is migrated and future accesses are now PILocalGetX requests, rather than PIRemoteGetX. If the miss does not cause a migration, then this case proceeds exactly as in DynPtr.

Local writeback. This case is identical to DynPtr. If a formerly remotely-allocated page has been replicated or migrated into the local memory of a node, then a writeback to any lines on that page will be a local writeback.

Remote writeback. This case is identical to DynPtr.

Note that the coherence protocol itself is unchanged, except for the code to implement cache miss counters and the processor-interrupt code. These changes are described in more detail in [VDG+96] and [Ver98].

3.6.2 Characterization in the design space

Where is data replicated/migrated? As in DynPtr, all data can be replicated in the processor caches. Replication/migration also occurs in memory: specifically, remote pages can be migrated/replicated locally, becoming local pages (in contrast to other schemes where they remain remote pages but misses are satisfied in the requesting node's memory).

Who decides when to replicate/migrate? The cache-coherence protocol replicates data in caches. The OS decides when to migrate/replicate into memory. The cache-based replication is *eager*, as in the other schemes–that is, a datum is replicated as soon as it is referenced. In contrast to the other schemes, the OS-based replication/migration is *delayed*–only after a certain number of misses have occurred to lines on a page is that page replicated or migrated.

What happens when a conflict occurs? Pages are migrated or replicated if free space exists on a node, so a replica does not displace other replicas, and the OS does not cause any additional conflicts in memory. Thus,

5. The OS uses a bcopy, which utilizes the underlying cache-coherence protocol to locate the desired lines and copy them to the appropriate region in memory.

the only conflicts that need to be dealt with are those in the processor caches, which are handled the same way as in DynPtr: a writeback is generated to the home, which sinks the data. The advantage of this approach is that for migrated/replicated pages the home and backing store are moved to the requestor nodes, so the writeback is satisfied locally.

What is the resulting view of memory? Part of memory is used for backing store for PP protocol code/data, including the directory information, and for cache miss counters. The rest of memory is allocatable by the operating system for application code/data, or can be used for replication/migration.

What are the potential benefits over base cache coherence? Misses to remote memory can be converted into local misses, enhancing data locality. Because the partitioning in memory is dynamic, potentially all locations on every node can be devoted to data that had previously been remote. Moreover, in protocols like RAC and COMA, migration and replication occur at the granularity of a cache line, but the addresses of the replicated/migrated lines remain remote addresses. References to such migrated/replicated lines invoke PIRemote handlers in those protocols. PIRemoteGet/GetX/Upgrade handlers in RAC/COMA require tag and state checks to determine if the data is in the requestor's memory. In contrast, in MigRep, the addresses are remapped locally, and are thus local addresses. References to such replicated/migrated data invoke PILocal handlers. The PILocalGet/GetX/Upgrade handlers are the same as that for DynPtr, and merely check a dirty bit and a local bit to determine if the data is in the local node's memory, without requiring any tag check. As a result, PILocal handlers are typically less expensive than PIRemote handlers if the data is on the requesting node. With MigRep, even if the same number of references is satisfied in the requestor's memory as in RAC or COMA, the references are satisfied with these less expensive PILocal handlers, thereby potentially providing performance gains over RAC or COMA.

What are the challenges in building this scheme vs. base cache coherence? The complexity in MigRep is shifted from the hardware/firmware to the software, and is discussed in detail in [Ver98]. Providing low overhead mechanisms in the OS is the bulk of the challenge. The sources of overhead, as described earlier, are: providing low-overhead counting code in the handlers, fielding the interrupt from the cache-miss counting code, making the policy decision for a page, allocating and copying the page, and maintaining page coherence through locks and the flushing of TLBs. Maintaining page coherence relies on tracking replicas and keeping some notion of master copies, similar to COMA. However, the resource constraints related to MAGIC queues and buffering seen by the lower-level COMA protocol are abstracted from the OS in MigRep.

In total, about 300 lines of code must be added to DynPtr to support the protocol-level modifications needed for cache-miss counting and interrupt handling in MigRep. No new handlers are required, although new cases are required in the handlers that perform communication between MAGIC and the OS. Of course, the complexity in MigRep stems mostly from the OS modifications required, rather than from the PP code mod-

Protocol	Memory used as a cache?	Memory cache HW/SW controlled?	Granularity of replication/ migration in memory	Memory statically partitioned for cache?
DynPtr	No	n/a	n/a	n/a
RAC	Yes	HW	cache line	yes
COMA-F	Yes	HW	cache line	no
MigRep	Yes	SW	page	no

TABLE 10. Qualitative comparison of replication/migration strategies. n/a means not applicable**.**

Protocol	# of “core” handlers	Handlers changed from DynPtr	# lines of handler code	OS changes relative to DynPtr OS	Implementation and debugging time required
DynPtr	65	n/a	8400	n/a	8 months
RAC	66	15	8900	10 lines	7 months
COMA-F	87	all	15400	10 lines	21 months
MigRep	65	9	8700	3600 lines	6 months

TABLE 11. Static complexity of replication/migration strategies.

ifications needed. Adding MigRep to the IRIX5.3 kernel requires the addition of about 3600 lines of code [Ver99]. The total static code size of IRIX5.3 is about 500,000 lines[Gov99].

3.7 Qualitative comparison of approaches

We have chosen a set of schemes that span the design space of replication/migration schemes, ranging from hardware-only mechanisms like base cache coherence to software-based schemes like OS-based page migration/replication. Hybrid hardware/software schemes will be considered in later chapters. Within this range, our protocols differ along various key axes. We summarize the qualitative differences between the schemes in Table 10, and also demonstrate the differences in complexity by showing some key protocol characteristics in Table 11. Of course, each of these schemes can be implemented on FLASH using the programmable memory controller and single pool of DRAM on each node. We depict a conceptual view of this flexible use of memory for each of our schemes in Figure 26. Notice that each scheme is essentially partitioning memory into a section for critical protocol code and data structures, a section for conventional memory, and a section for cache, as shown in a more simplified manner in Figure 3 of Chapter 1.

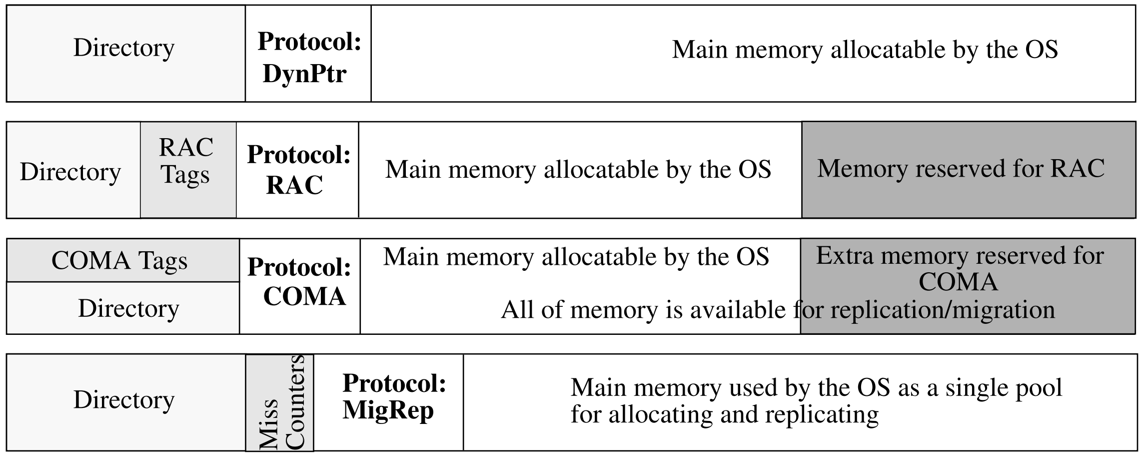


FIGURE 26. The use of local memory on each node by the different protocols. The DRAM on a node is used for directory storage, tags, counters, protocol code, and caching of remote data, in addition to main memory that is visible to the OS**.** The picture is not to scale.

Table 11 also illustrates the effort involved in implementing each of the schemes. The base DynPtr prototype scheme required about eight months to implement and debug[Hei93]. Numerous additions and feature changes have been ongoing. Building upon the already existing DynPtr design, the prototype RAC protocol required approximately one month to specify, one week to implement, and six months to fully debug. While it seems a rather disproportionate amount of time is spent debugging, there are numerous corner cases that arise that take significant amounts of time to track down. This time does not include other additions and feature changes, like testing a write-through vs. writeback RAC. The COMA protocol was implemented before the RAC protocol, and did not build upon DynPtr. It took roughly three months to specify the COMA protocol, six months to implement, and a year to debug. The long length of time required for COMA can be attributed to numerous factors. First, learning the protocol tools and environment required significant effort, especially since the author of COMA was different from the author of DynPtr. Previous experience with these tools would have significantly reduced implementation effort. Second, there are more resource checks in COMA than in DynPtr, leading to more code and more debugging effort. Third, a number of COMA issues stress parts of the system that are not stressed by DynPtr. For example, base DynPtr has fewer deadlock problems than COMA. The solution to these DynPtr deadlock issues involves two uses of the software queue in the PP–namely, generating invalidations that could not be sent when a write request was originally received, and one case of a reply that causes another reply. COMA's numerous deadlock cases require using the software queue for a number of additional purposes, like storing data on replacements or storing data while waiting for invalidation acknowledgments to return. Debugging these aspects of the system, given they had not been exercised in this manner before, required significant time. Finally, there are numerous other transients and race conditions because of the increased interactions caused by the increased number of messages in the system, leading to more involved coding and debugging than in the other schemes. In contrast to all of the hardware-centric approaches, MigRep's effort was concentrated in the OS. MigRep required about one month to implement a prototype, and about five more months to get a complete version running.

3.8 Summary

By allowing the flexible use of memory, the FLASH machine allows us to implement the schemes we will evaluate in this thesis:

- The baseline approach is CC-NUMA. This scheme does not perform any memory caching. It requires some memory to store protocol code and data to allow communication of data among the processor caches. The remainder of memory is used for normal backing store.
- Our next scheme is a simple RAC. Part of memory is used for the memory cache, part of memory is used for application code/data and protocol code/data, and the rest is needed for tags and state on the RAC.
- Our third scheme is a COMA design. COMA can be implemented by storing tags and state on all of memory. Memory also stores protocol code/data and application code/data. The regions that are used for protocol code/data or for tags are stored in main memory, but are prevented from being migrated or replicated by being mapped in a special region of memory.
- Our final approach is page migration/replication (MigRep). A MigRep design can be implemented by storing miss counters in memory. When these miss counters overflow, MAGIC interrupts the main processor and the OS makes the policy decision on the page. A portion of memory is devoted to the protocol code/data, and the remainder can be used by the OS either for backing store, or for migrating and replicating pages.
- In subsequent chapters we will consider hybrid designs. A hybrid design borrows elements from each of the above schemes. Some memory is needed for protocol code and data, while some is needed for backing store. The remainder is used for counters, tags, state, and cache, as needed.

Implementation of these schemes on other architectures is typically not possible because protocols are hardwired at design time. Moreover, even if a machine has a flexible protocol engine, the inability to use memory flexibly may limit the types of protocols that can be used. For example, if there is no way for a portion of memory to be designated for use as a cache, then the machine is limited to cache-coherence protocols in which data is replicated only in hardware caches, rather than protocols that employ caching in memory.

The alternate protocols to DynPtr attempt to reduce application runtime by decreasing the memory stall component of execution time via replication/migration of data to local memory. The potential benefits from the different techniques depends on the kinds of misses a given workload exhibits. The two main kinds of misses are capacity/conflict misses, which arise because of the limited size and associativity of the processor caches, and coherence misses, which arise when data is actively being read and updated by multiple processors.

When capacity/conflict misses dominate, all of the schemes can help satisfy capacity/conflict misses locally by caching in memory or moving the data to the local node through migration or replication. MigRep will achieve relatively less locality because it counts misses and waits for a page to get hot, and does not cache eagerly. It waits for a consistent miss pattern across an entire page to emerge before deciding to migrate or replicate that page. Between COMA and RAC, the big difference is that COMA can dedicate all of the local memory to caching remote data (e.g., if the working set is large and we start with very poor initial allocation). In contrast, RAC can dedicate only as much memory as is statically allocated to it, which will likely be only a small fraction of the local node memory. In addition, COMA does not require copies of data to stay at nodes that are not actively accessing that line, as occurs in RAC with memory that is locally allocated, but is never used by the node on which the page is allocated.

When coherence misses dominate, data suffering from coherence misses is unlikely to benefit from any of the locality schemes because of frequent invalidations of cached copies. MigRep can be robust to such misses because the cache-miss counting and the policy together detect these patterns and take no action for such pages. The RAC scheme has a slightly higher handler overhead compared to DynPtr, so it will see some degradation. As compared to DynPtr and RAC, COMA has extremely high handler latencies and occupancies for coherence misses (as described in Section 3.5.3.3 when discussing write misses in COMA) and will perform poorly if coherence misses dominate.

Chapter 4
Quantitative Comparison of Replication/Migration Schemes

The previous chapter described the various schemes for replication/migration that we consider in this thesis. In this chapter we evaluate these schemes on a platform similar to the Stanford FLASH machine. We first describe the experimental environment and workloads that we use in our study. We then present performance results for some baseline parameters, and then a perform a sensitivity analysis on some of the key baseline parameters.

4.1 Experimental Environment

4.1.1 Machine Assumptions

We model our CC-NUMA machine based on the Stanford FLASH architecture[KOH+94]. The FLASH machine was not yet available at the time of this research, so we simulated our experiments using SimOS[RHW+95][RBD+97]. SimOS is a complete and accurate simulator of FLASH. SimOS is capable of booting a commercial operating system, Silicon Graphics' IRIX5.3 in this case, and executing any application that is binary compatible with IRIX. In conjunction with FlashLite, the threads-based memory system simulator for FLASH, SimOS accurately models the processors, caches, memory system, and I/O devices (disks, ethernet, etc.) of the system, including all the functions of the MAGIC chip. FlashLite simulates the same compiled cache-coherence protocol handlers used on the actual FLASH machine, using a cycle-accurate instruction set emulator as its protocol processor thread. The use of SimOS allows us to gather detailed statistics that might not have been available on real hardware, and also allows us to vary the design space in ways that would have been infeasible in hardware.

We model an eight processor FLASH machine. The benefits of migration and replication should be apparent even with this small configuration because the probability that a process would randomly find an address in local memory is already quite small (0.125). The following lists the other machine characteristics we assume: 300MHz processors with a TLB size of 64 entries; sequential consistency with blocking reads and writes[1]; separate 32KB two-way set-associative first-level I and D caches with a one cycle hit time; a unified 512KB two-way set-associative second-level cache with a 50ns hit time. For the Raytrace and Splash workloads (described in the next section) we use a 256KB cache because their working sets are smaller. The base configuration has 256MB of memory, i.e. 32MB per node. MAGIC's processor clock speed is 100 MHz. The MAGIC ICache is assumed[2] to be 64KB and the MAGIC DCache is assumed to be 1MB. For the DynPtr protocol, assuming no contention and perfect MAGIC cache behavior, the local miss latency is 190ns and the remote read miss latency for data clean at the home[3] is 864ns. FlashLite, of course, properly accounts for contention at all its interfaces and at the router nodes and accurately models the MAGIC caches, so actual miss times may be greater unless the machine is truly idle.

The SGI IRIX5.3 kernel has some significant bottlenecks even at the eight processor level. All of the kernel code and data is allocated in low memory and so ends up on node 0. There is a coarse lock, `memory_lock`, for most of the operations related to the VM system. This lock is the source of much contention for workloads that are kernel-intensive. For the MigRep scheme IRIX5.3 was modified[VDG+96][Ver98] to implement the migration and replication of pages.

4.1.2 Workload Characterization

The value of a study such as this depends critically on the workloads used. We use four diverse and realistic workloads to capture some of the major uses of compute servers. These workloads are summarized in Table 12. Table 13 shows the breakdown of execution time for the workloads when run with the base DynPtr protocol, and Table 14 shows the miss characterizations. We divide misses into user, kernel, and synchronization (sync) misses, and list percentages of capacity/conflict, cold, and coherence misses for instructions and data.

We will next discuss the four workloads–Raytrace, Splash, Engineering, and Pmake–in more detail.

---

1. The SGI IRIX kernel assumes the sequential consistency memory model.
2. While the value we use is larger than in the real machine, this size allows us to focus on the performance of the protocol enhancements themselves, rather than having results potentially dominated by ICache miss rates.
3. For an eight node hypercube, the average number of hops traversed by a message is 2.1. The computed remote latency assumes this number of hops to reach the home.

Name	Contents	Notes
Raytrace	Raytrace	parallel graphics applications (rendering a scene)
Splash	Raytrace & Ocean	multiprogrammed, compute-intensive parallel applications
Engineering	6 Flashlite, 6 Verilog	multiprogrammed, compute-intensive serial applications
Pmake	4 four-way parallel Makes	software development (compilation of gnuchess)

TABLE 12. Description of the workloads. All workloads are run on an eight processor machine.

Single Parallel Application (Raytrace): This workload consists of Raytrace[WOT+95], a single compute-intensive parallel graphics algorithm widely used for rendering images. A 3-dimensional scene is rendered onto a 2D image plane using the raytrace algorithm. In the raytrace algorithm, a ray representing a light source is shot through a pixel in a 2D image plane and bounces off objects in the 3D scene according to the objects' opacities. Combining the information about which objects cause reflection or refraction of light allows the intensity and color of the individual pixel to be determined. The eight processes are locked to individual processors, a common practice for dedicated-use workloads. This workload has very little kernel activity. Memory stall time is significant, about 33% of the non-idle execution time, with the large part spent in user stall time. Most data misses in Raytrace are to a large read-only shared data structure representing the scene to be rendered. This data structure overflows the secondary cache and leads to a large number of capacity misses, clearly demonstrated in Table 14. Therefore, this workload can potentially benefit from improved data locality. This workload is not load-balanced by the OS, so there is no issue of process migrations disturbing data locality, and one might imagine statically partitioning the data to provide good locality and load balance. However, the unstructured accesses to the main data structure prevent effective static partitioning. These unstructured accesses are an artifact of the raytrace algorithm, because the ray bounces off objects in an unpredictable way. The outcome is dependent on scene geometry, so the user cannot determine beforehand precisely which pixels will be influenced by which objects.

Multiprogrammed Scientific Workload (Splash): The second workload consists of parallel invocations of Raytrace and Ocean[WOT+95]. Ocean is a multigrid solver that simulates boundary and eddy currents in an ocean basin. The basin is discretized into a grid, with numerous arrays storing forces and potential at the grid points. The applications enter and leave the system at different times, and a space-partitioning approach, similar to scheduler-activations[ABL+91][TuG91], is used for scheduling the jobs. Both Ocean and Raytrace have large datasets that overflow the secondary cache. In the process of performing the multigrid computation, Ocean exhibits nearest-neighbor communication, and also incurs numerous misses to large private data arrays. Thus, we expect to improve locality by migrating data. As described earlier, Raytrace suffers from misses to a large, shared read-only data structure. About 41% of the non-idle execution time is memory

Workload	Cumulative CPU Time (sec)	CPU Time Breakdown (%)			Stall Time (% Non-Idle)			
					Kernel		User	
		User	Kern	Idle	Instr.	Data	Instr.	Data
Raytrace	30.52	75	8	17	1.9	3.9	5.4	21.3
Splash	41.67	64	11	25	3.1	8.4	3.4	26.5
Engineering	37.43	78	6	16	1.4	3.8	33.1	29.2
Pmake	30.84	21	41	38	5.3	43.3	2.8	5.3

TABLE 13. Execution time breakdown of the workloads under DynPtr.

Workload	Mode	# misses (millions)	% of total number of misses	Instructions		Data		
				% cold	% cap./ conf.	% cold	% cap/conf.	% coher.
Raytrace	User	6.40	88.9	0.06	17.7	4.74	77.0	0.51
	Kernel	0.80	11.1	1.06	31.5	3.60	54.2	9.66
Splash	User	7.76	83.9	0.13	13.3	5.28	80.2	1.15
	Kernel	1.49	16.1	0.76	39.1	2.45	38.4	19.3
Engineering	User	12.0	94.2	1.17	62.1	1.13	35.5	0.09
	Kernel	0.75	5.8	1.32	35.2	6.69	38.8	18.0
Pmake	User	1.03	31.0	6.07	40.7	8.11	44.9	0.19
	Kernel	2.30	69.0	0.73	24.8	5.71	19.0	49.8

TABLE 14. Miss Characterization of the workloads.

stall time. This workload should benefit from better data locality because the misses are predominantly capacity misses, but about 20% of the kernel misses are coherence misses.

Multiprogrammed Engineering Workload (Engineering): Engineering consists of large, memory-intensive, uniprocessor applications. This is a multiprogrammed workload, scheduled by UNIX priority scheduling with affinity[TuG91]. The workload consists of copies of two applications. One is the commercial Verilog simulator VCS, simulating a large VLSI circuit, in this case, a snapshot of the FLASH MAGIC chip. VCS compiles the simulated circuit into C code, and the resulting large code segment causes a high user instruction stall time. The other application is FlashLite, the memory system simulator of the FLASH machine. This is an extremely memory-intensive workload, with about 67% of the non-idle execution time spent stalled for memory, almost all of which is user memory stall time. Over 95% of the user misses are capacity misses, indicating excellent potential improvements from better data locality.

Multiprogrammed Software Development (Pmake): Our final workload consists of four Pmake jobs, each compiling different files from the gnuchess program with four-way parallelism. The workload is I/Ointensive, with a lot of system activity from many small short-lived processes, such as compilers and linkers. Each job has its source on a separate disk, and the file buffer cache is warm with all the necessary files. UNIX priority scheduling with affinity is used. The kernel time is double the user time in this workload. Kernel instruction and data references, rather than user references, account for the bulk of the memory stall time; kernel stall time is about 50% and user stall time only 8.1% of non-idle execution time. 85% of user misses (instruction and data) and 44% of kernel misses are capacity misses, but almost 50% of the kernel misses are coherence misses. While the benefits from data locality are not obvious for the Pmake workload, we use Pmake for two reasons: first, to help focus on the migration and replication potential in the kernel; second, to understand the robustness of our schemes in the face of coherence traffic (for which they are not explicitly designed to perform well).

4.2 Simulation Results

The goal of our study is to determine how to best use the pool of local memory on a node in a CC-NUMA system to improve data locality and therefore application performance. For our first comparison, we start with the assumption that there is sufficient memory for the following purposes: the OS is able to fit the footprint of the application in main memory, there is enough reserved memory for caching in RAC and COMA, there is sufficient main memory for MigRep to replicate pages, and there is enough memory available for any protocol memory overheads. For our workloads this translates to 32MB per node. RAC and COMA reserve 16MB per node for local caching[4].

For each workload, we first show graphs of the execution time (Figures 27 and 28) of each of the data locality schemes normalized to that of DynPtr[5]. Execution time is broken down into user instructions, user local and remote stall, kernel instructions, kernel local and remote stall, and kernel synchronization time (time spent spin-waiting for locks). For each workload, Table 15 provides more detailed data showing the handler costs, PP occupancy, and percentage of misses satisfied from local memory (Local %).

4.2.1 Results for the Raytrace workload

Raytrace is a dedicated-use workload. Although the jobs are not moving around in the system, the demands on data migration and replication are still high because of the unstructured data accesses to the 3D image

4. Given the direct-mapped AM in our COMA implementation, the amount of memory needed to avoid AM conflicts would be unreasonably large. Therefore, COMA will suffer some conflicts.

5. The kernel used for MigRep is different from that used for the other runs. The difference in kernel time between the DynPtr kernel and the MigRep kernel (with migration and replication disabled) is practically zero for all of the workloads except Pmake. For Pmake, the runs with the DynPtr kernel are 21% faster than the runs with the MigRep kernel with migration and replication disabled. The bulk of the increase is attributable to synchronization for the single coarse lock `memory_lock` in IRIX5.3, and the allocation of kernel code and data in the memory of node 0 causing great contention on this node (described in Section 4.1). The characteristics of Pmake, forking and exiting of many short-lived jobs, bring out the worst of this problem. For Pmake, to clearly show the effects of migration and replication, we normalize the MigRep numbers to that of the MigRep kernel with migration and replication disabled.

data. The majority of misses are user-level misses, primarily to the 3D image data structure. 89% of the misses are user-level misses, and almost 95% of these misses are capacity/conflict misses. As a result, we expect each of our schemes to be able to improve performance. The key determinant of performance is whether the locality benefits outweigh the added complexity of the given scheme.

As seen in Figure 27, the RAC protocol performs significantly better than DynPtr (27% faster[6]). Although RAC is higher overhead than DynPtr, as seen by the latencies in Table 5 of Chapter 3, the RAC improves locality sufficiently to outweigh this overhead. We can see the locality versus complexity trade-off by considering Table 15. RAC dramatically improves locality relative to DynPtr: 94% of references are satisfied locally in RAC, while only 14% are satisfied locally in DynPtr. As a result, fewer handlers are run in RAC (2.18 handler calls per miss vs. 4.55 in DynPtr). In addition, although RAC remote handlers are more expensive than their DynPtr counterparts, the large number of RAC hits mean that the majority of handlers running in RAC are simply PIRemoteGet or PIRemoteGetX handlers. Thus, the most costly handlers, NIRemotePut and NIRemotePutX, are run relatively infrequently. As a result, the average occupancy per handler approaches that of the occupancy of PIRemoteGet or PIRemoteGetX. An additional benefit of RAC is that the average utilization of the PPs in the system is reduced relative to DynPtr, because each node performs relatively less work than in DynPtr. A RAC hit causes a reference to be satisfied by one handler, which takes up less aggregate PP time than references satisfied remotely and requiring up to four handlers, one at the requestor, one at the home, one at a remote node, and the final reply handler at the requestor. A final benefit of the RAC is that it can reduce hotspotting, because references are not constantly going to the home to get a datum: instead, each can store a copy at its local node. As a result, the utilization of the busiest PP is dramatically reduced in RAC relative to DynPtr.

MigRep and COMA also improve performance compared to DynPtr. MigRep provides relatively less locality benefits than RAC because of its delayed caching, while COMA suffers because of conflicts between local and remote data in a given node's AM. Both MigRep and COMA improve locality sufficiently to reduce the number of handlers that run per miss (3.08) relative to DynPtr (4.55). This workload gives an indication of the cost of COMA's complexity. Despite COMA reducing the number of handler calls per miss, its handlers are complex enough that the average occupancy per handler is quite a bit higher than DynPtr (25.3 vs. 15.6 for DynPtr). The handlers run for replacements are not on the latency path of data retrieval, but add occupancy after a datum has been received. As a result, and also because of the extra occupancy of basic request handlers, the average PP utilization increases relative to DynPtr. The majority of MigRep handlers, on the other hand, are identical to those of DynPtr, and the overheads of the handlers that differ in MigRep are small relative to DynPtr. MigRep is therefore not hurt by excessive PP costs relative to DynPtr.

6. The figures show normalized execution time. Speedup is computed based on speed, which is the inverse of execution time[PaH96].

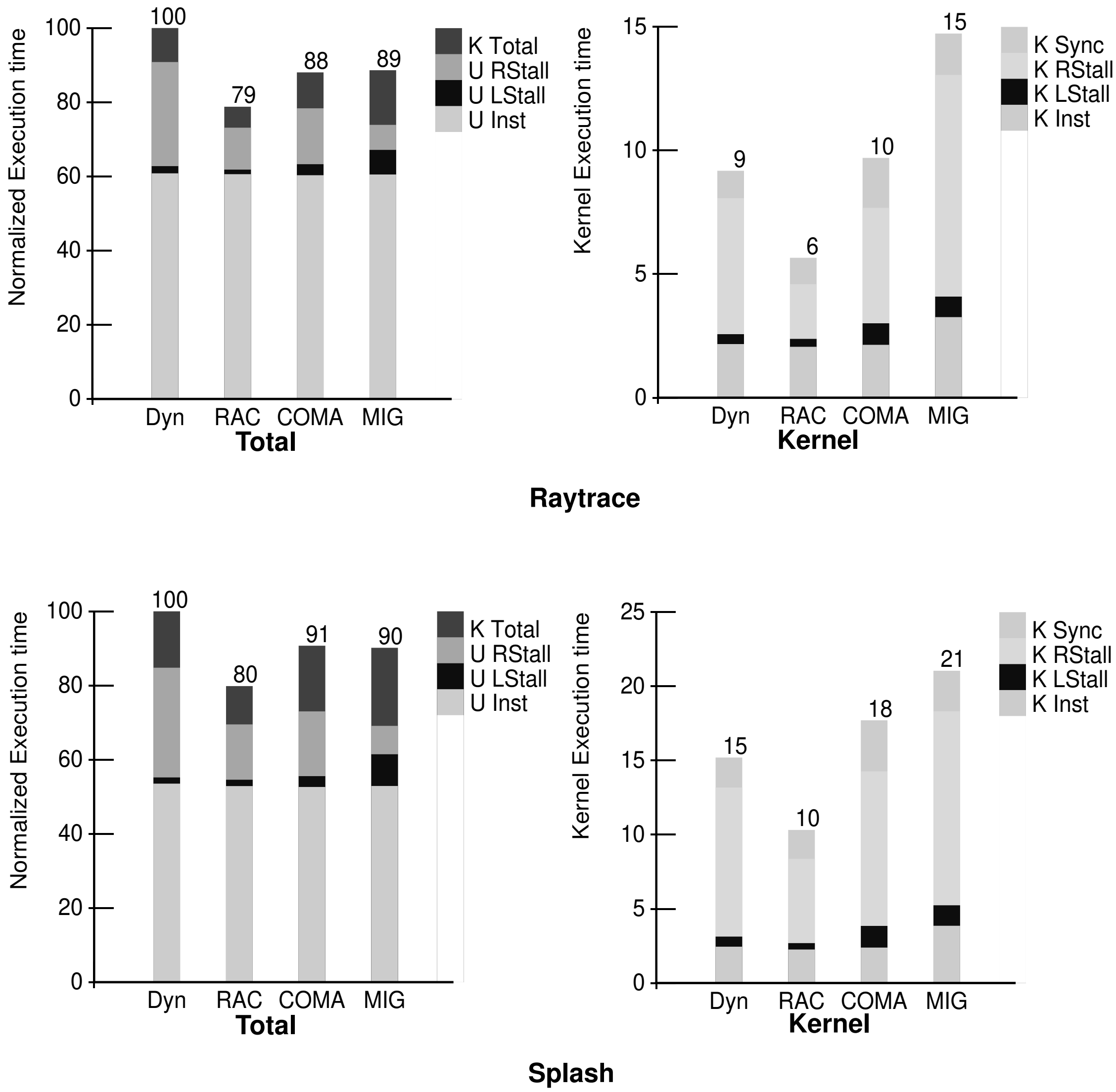


FIGURE 27. Base execution time comparison, Raytrace and Splash. For each workload we show the total execution time normalized to DynPtr in the graph on the left, and a more detailed breakdown of the kernel component of the execution time (K Total) in the graph on the right.

Instead, additional kernel time to implement the migration/replication policies adds overhead and degrades some of its gains.

4.2.2 Results for the Splash workload

Unlike the Raytrace workload, Splash contains more processes than processors. As a result, processes will be exiting and entering the system intermittently, taxing the data locality mechanisms we provide. The individual raytrace application in Splash should perform the same as in the Raytrace workload. Ocean should add some capacity misses as well, which can be captured by migration since a given datum within the grid is used

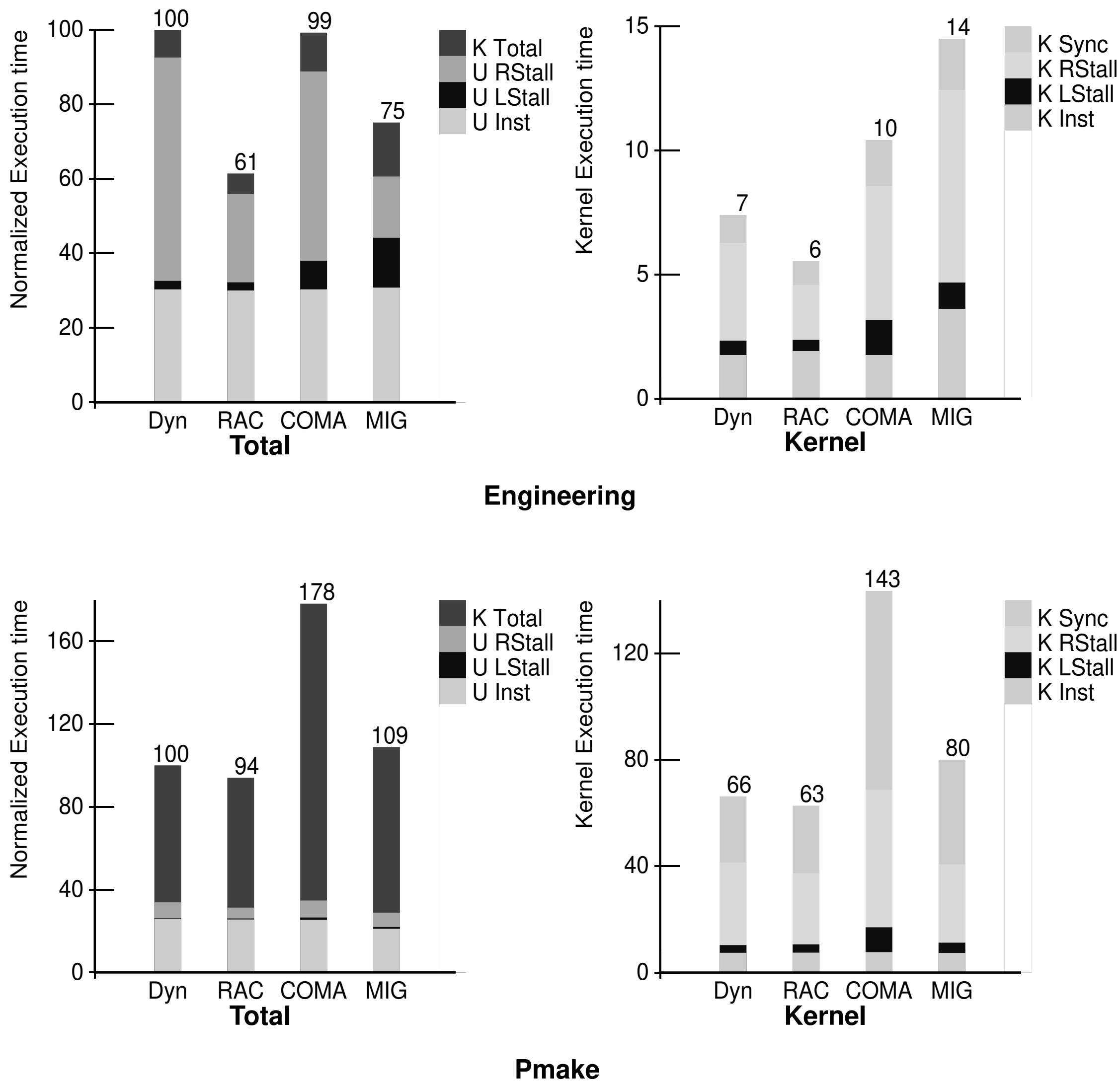


FIGURE 28. Base results, Engineering and Pmake. Total normalized execution time is shown in the graph on the left, and a detailed breakdown of the kernel component of execution time is shown on the right. Engineering execution time is normalized to DynPtr, while Pmake is normalized to that of the MigRep kernel with migration and replication disabled**.**

mostly by one processor. The miss profile in Splash is generally very similar to Raytrace, seen in Table 14, so the results we see are similar. As Figure 27 demonstrates, RAC performs significantly better than DynPtr (25% faster). COMA and MigRep show almost the same performance increase versus DynPtr (about 12-14% faster than DynPtr), but neither performs as well as RAC. Compared to MigRep, COMA has a slightly larger user time and a slightly smaller kernel time because COMA is able to migrate and replicate kernel code and data, while MigRep cannot.

The improvements in locality and the subsequent impact on handler calls per miss, occupancy per handler, and utilization of PPs is very similar to the analogous results in Raytrace, as seen in Table 15. However, because the kernel must spend more time scheduling jobs, the kernel coherence traffic increases relative to Raytrace (see Table 14). The locality improvements are thus somewhat less than in Raytrace, and the overall handler costs increase accordingly.

4.2.3 Results for the Engineering workload

The Engineering workload presents a different picture. In this workload the RAC and MigRep schemes both show large gains, 64% and 33% faster than DynPtr, respectively. This is a memory-intensive workload with a high miss rate and consequently a higher average controller utilization. While Engineering has a large instruction footprint, making it amenable to gains, it also has more processes than processors, so the kernel must get involved scheduling processes. Because most kernel data structures are allocated out of node 0, it becomes a hot spot, and has high utilization, even in a scheme like RAC. The higher average and peak controller utilization for RAC consequently overshadows some of the gains due to increased locality.

The large instruction footprint in Engineering is particularly suitable for MigRep, since MigRep is able to replicate or migrate entire pages of read-only code. MigRep suffers somewhat from its delayed caching and from its inability to replicate kernel code and data, providing lower data locality than RAC. In addition, MigRep adds kernel time because of the migration/replication code. However, the low controller overhead for MigRep helps mitigate some of the decreased locality and increased kernel time, bringing its performance relatively close to that of RAC.

In contrast to RAC and MigRep, COMA is completely occupancy-bound, and the occupancy effects for this high miss-rate workload completely nullify any gains from better locality. The problem again involves conflicts between local and remote data that wish to occupy the same slot in the AM. In addition, contention at node 0 and frequent use of expensive replacement handlers (due to the local/remote conflicts) cause the occupancy of COMA to increase dramatically relative to DynPtr (28.6 cycles per handler in COMA versus 13.3 in DynPtr). The resulting utilization of the PPs in COMA is much worse than DynPtr (44.9% vs. 28.1%). The severe hot spotting and frequent invocation of replacement handlers at node 0 is seen in the utilization of the busiest PP (79.2% for COMA vs. 70.7% for DynPtr).

4.2.4 Results for the Pmake workload

The Pmake workload is different from the previous three workloads for two main reasons. First, it spends a larger fraction of its time in kernel mode than in user mode because of the large number of processes requiring scheduling by the OS. Second, it consists of many small jobs rather than a few larger ones, so a large fraction of its misses are coherence misses, not capacity. Most of the kernel misses go to node 0, since the kernel is allocated out of node 0 in IRIX5.3, leading to extremely high controller utilization on that node.

Protocol	Local %	Handler calls/ miss	Avg. occupancy/ handler (PP cycles)	Avg. utilization of PPs(%)	Utilization of busiest PP (%)
Raytrace					
DynPtr	14	4.55	15.6	18.2	58.8
RAC	94	2.18	18.7	13.7	16.2
COMA	81	3.08	25.3	23.1	40.0
MigRep	63	3.08	12.8	13.5	38.1
Splash					
DynPtr	13	4.53	13.8	17.9	38.1
RAC	90	2.37	20.8	17.1	26.7
COMA	82	3.04	25.8	23.5	45.5
MigRep	62	3.09	13.1	14.4	38.2
Engineering					
DynPtr	10	4.37	13.3	28.1	70.7
RAC	94	2.04	19.8	33.9	44.7
COMA	71	3.28	28.6	44.9	79.2
MigRep	66	2.80	11.9	24.6	50.7
Pmake					
DynPtr	13	4.69	15.3	14.6	69.5
RAC	57	3.78	26.8	20.2	58.2
COMA	56	4.98	33.8	22.4	82.7
MigRep	25	4.51	16.3	20.5	80.6

TABLE 15. Handler and occupancy statistics for the workloads. The average occupancy per handler is computed by dividing the total number of occupancy cycles by the number of handler invocations. Local % is the percentage of references satisfied by local memory**.**

The RAC provides about 6% improvement through eager caching of user and kernel data. MigRep is unable to deal with kernel misses, and cannot provide any benefits for small jobs that finish quickly. Therefore, it is unable to significantly improve data locality, and DynPtr runs about 9% faster.

The kernel misses are detrimental for RAC and MigRep, but are particularly damaging for COMA. Coherence (write) misses are very expensive in COMA, as discussed in the previous chapter, because the home has to retrieve the data from its current owner, send out invalidations to all sharers, and then hold on to the data until all the invalidations are received, in order to enforce sequential consistency. Requiring retrieval of data from the current owner eliminates the possible optimization of allowing the home to satisfy the request if it has a non-master copy of the datum. In addition, the handler routines for storing and retrieving data from memory when all invalidations have returned are quite expensive. When a datum is received at the home before invalidations are returned, it is enqueued on the software queue. This is an expensive operation since the queueing instructions are a sequence of loads and stores, presumably to locations that are infrequently used, and therefore not in the MAGIC data cache. As a result, these queueing instructions incur high latency. Retrieving the data from the software queue when all invalidations have returned is similarly expensive. Recall that in DynPtr, if the datum were shared, then the home would already have a copy, and would only need to send invalidations. There would be no need to store the datum in any other storage area. As a result of the extra handlers required to store and retrieve the data, and the need to retrieve data from its current owner, the number of handlers per miss increases (4.98 vs. 4.69 for DynPtr) despite the better locality (56% accesses satisfied locally vs. 13% for DynPtr). The resulting average occupancy per miss is 2.2 times that of DynPtr. This large increase in occupancy results in DynPtr running 78% faster than COMA, entirely accounted for by COMA's increase in kernel stall time, since the kernel is the source of the coherence misses.

4.2.5 Summary of base results

The observed performance of the data locality schemes is fundamentally dependent on two factors: the improvement in data locality that the scheme is able to provide, and the real cost of providing the better locality (implemented handler complexity or operating system overhead). The following trends are characteristic of the different schemes across the workloads:

- RAC, COMA, and MigRep are all able to improve the data locality seen by the workloads compared to DynPtr. In Raytrace, Splash, and Engineering, RAC provides 90% locality versus 10-12% for DynPtr, and MigRep and COMA provide between 60-80% locality. RAC and COMA are able to achieve a better locality than MigRep because they perform eager caching while MigRep waits for hot pages. They are also able to extract locality at a finer grain and to improve the locality of kernel data, while MigRep is unable to replicate kernel code and data and therefore cannot reduce kernel stall time. COMA achieves a lower data locality than RAC, because COMA needs to find a home for the master copies when replacements occur, even at the cost of throwing away replicas. Although not shown in the table, the kernel locality for RAC varies between 46-86% for our workloads, and the kernel locality for COMA is 50-72%. MigRep does not benefit the kernel, but has a slightly different allocation policy, resulting in a base of 22-27% locality. In contrast, DynPtr has 14-20% kernel locality.
- Locally-satisfied cache misses do not require remote intervention, so the improved data locality for all schemes results in a smaller number of handlers invoked per miss compared to DynPtr. DynPtr executes between 4 and 5 handler calls per miss, while RAC, COMA, and MigRep typically execute between 2 and 3 handler calls per miss. The exception is Pmake, in which RAC and MigRep still execute fewer handlers from DynPtr (3.78 and 4.51 for RAC and MigRep, respectively, vs. 4.69 for DynPtr), while increased coherence traffic causes COMA to execute more handlers than DynPtr.
- The significantly higher handler complexity of COMA results in a much higher average handler occu-

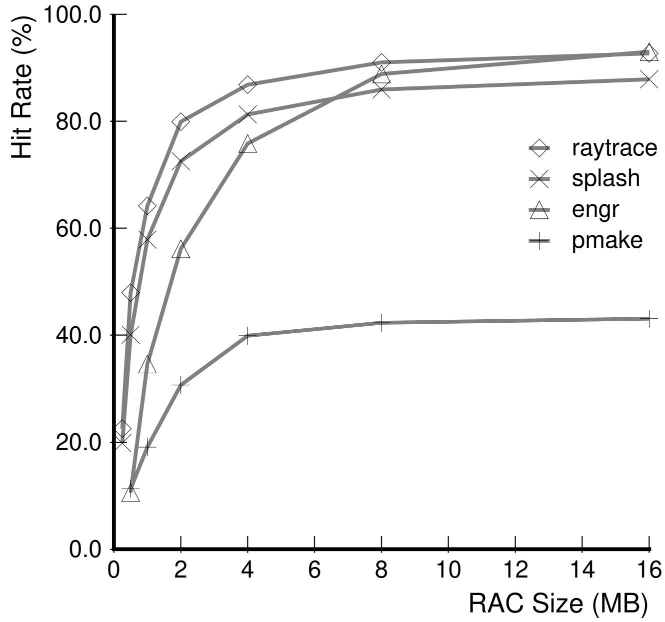


FIGURE 29. Effect of RAC size on RAC hit rate.

pancy, which overshadows its gains in locality relative to DynPtr. The average utilization for DynPtr is typically between 15-30%, while for COMA it varies from 20-40%.

- The broad trends are similar for kernel and user time. MigRep cannot improve locality for kernel misses because it does not migrate or replicate kernel pages. In addition, MigRep incurs kernel overhead to migrate and replicate user pages. However, the low controller occupancy for MigRep helps mitigate some of the loss in locality.

Overall, we see the fundamental performance trade-off between better data locality and increased handler occupancy across all the workloads, whether they are user-intensive or kernel-intensive. With no memory constraints, RAC performs best, because it is able to improve data locality with only a small increase in handler complexity over DynPtr. While the trade-off between better locality and protocol complexity is true in general, it is more extreme in the FLASH controller where we can see the effects of handler complexity in our implementation of COMA. COMA performs worse than MigRep because COMA has much greater handler complexity, even though MigRep achieves poorer data locality and has associated kernel overheads. However, COMA is able to improve locality and provide benefits in some cases.

4.2.6 Exploration of Parameters

The two results that stand out from the experiments in the previous section are that the RAC consistently performs well and that COMA suffers because of handler complexity and the resulting high controller occu-

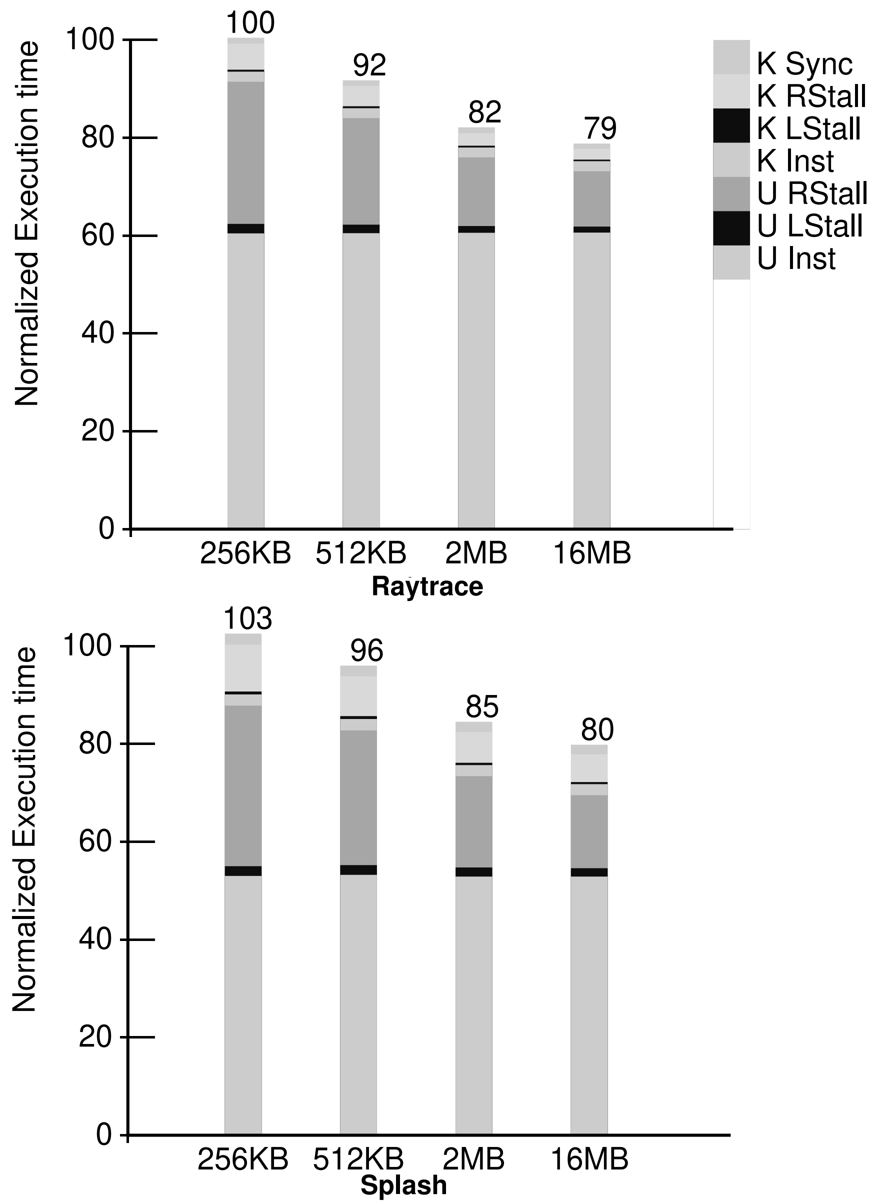


FIGURE 30. Effect of changing RAC size, Raytrace and Splash. All times are normalized to the equivalent DynPtr run. The working sets of Raytrace and Splash are smaller than for Pmake and Engineering, so we use smaller RAC sizes.

pancy. This section explores these two points more closely by considering the effect of RAC size on the performance of RAC, and the effect of controller speed on the performance of COMA.

4.2.6.1 RAC Size

The experiments for the base results assumed that RAC had 16MB of memory per node reserved for caching data. Memory that is local to a node is split between the RAC and the portion available for allocation by the OS. Therefore, any memory assigned to the RAC cannot be used by the OS to allocate pages. A large stati-

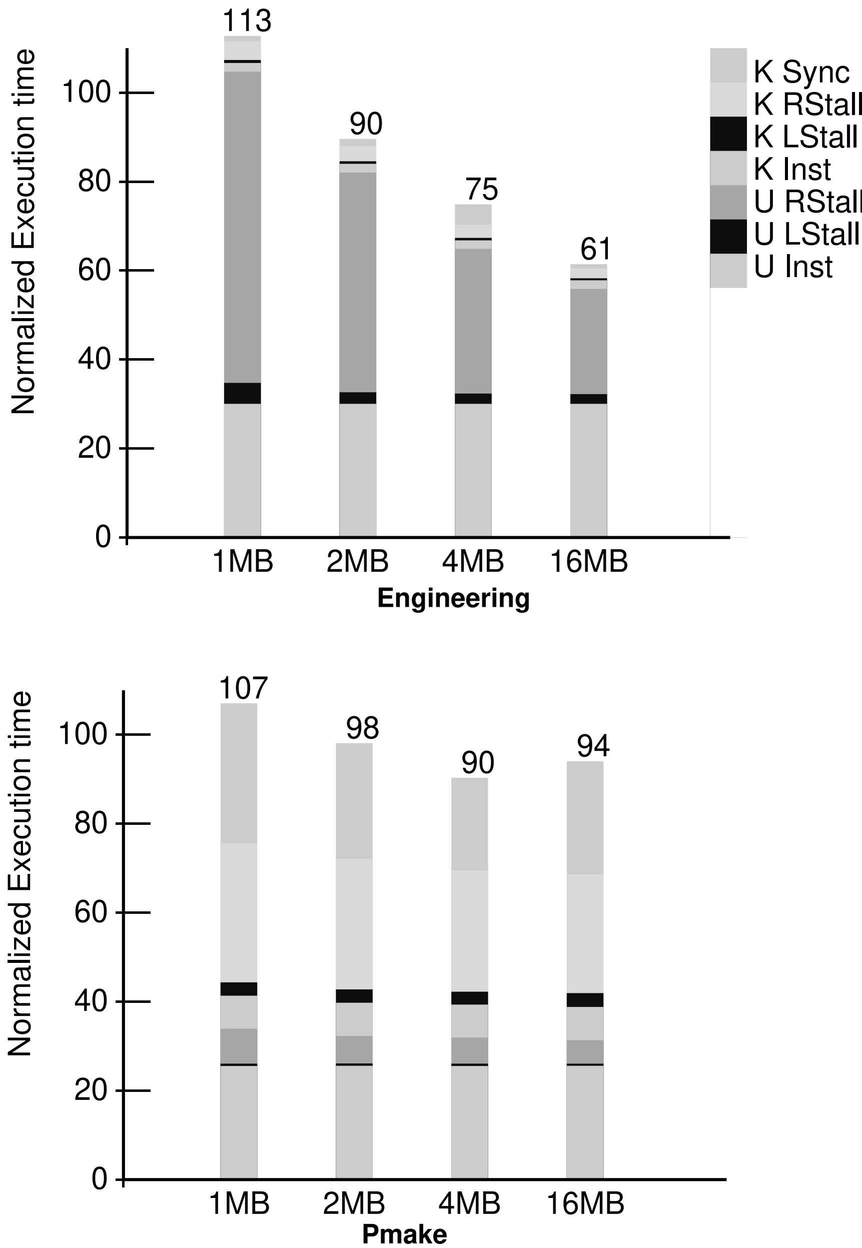


FIGURE 31. Effect of changing RAC size, Engineering and Pmake. All times are normalized to the equivalent DynPtr run**.**

cally-sized RAC means less memory available for the OS, and this may potentially cause unnecessary paging or may limit the workload size. Given a more limited RAC size, one cannot assume that the working set will always fit in the RAC. It is important to understand the limitations of the RAC design and its performance in the regime where working sets do not easily fit in the RAC.

To measure the effect of memory pressure on the RAC protocol, we varied the size of the RAC from 256KB to 16MB for each application. Figure 29 shows the hit rate in the RAC for the different RAC sizes. Figure 30 shows the execution time for Raytrace and Splash normalized to DynPtr, and Figure 31 does the same for Engineering and Pmake.

As expected, the performance of the RAC scheme drops as the RAC size is reduced. Figure 29 clearly shows that as the RAC size is reduced, the hit rate in the RAC drops sharply for all the applications (with the knee around 2-4MB). The results show that at smaller RAC sizes most of the gains over DynPtr are lost. Interestingly, the Engineering workload and Pmake show RAC performance that is worse than that of DynPtr for smaller RAC sizes, as seen in Figure 31. The increase is in user time for the Engineering workload and primarily in kernel time for the Pmake workload. This arises because of the larger handler occupancy of RAC along with the reduced data locality at the smaller RAC size.

Another surprising result from Figure 31 is that too large a RAC can be detrimental to performance. In Pmake, the performance with a RAC size of 4MB is actually better than performance with a 16MB RAC. This occurs because of increased cache-to-cache sharing with a large RAC. With a 16MB RAC, certain coherence data stays on a remote node, prompting accesses to that line to go to the remote node. A 4MB RAC is small enough that such data is actually evicted from the RAC and written back to the home, so that subsequent accesses are satisfied at the home and do not require remote accesses. Another option for alleviating this problem with large RACs is to employ a write-through RAC, rather than a writeback RAC. In a write-through RAC, the RAC never keeps the last copy of a line: if a node evicts a dirty line from its processor cache, that datum is written to the RAC in a shared state, and also written back to the home. Future read accesses to that line can be satisfied by the home. Preliminary experiments with such an approach indicate that it successfully alleviates the cache-to-cache sharing problem. The write-through RAC, however, has the drawback that it can generate large amounts of writeback traffic to the home if the processor cache frequently evicts dirty lines. To avoid excessive traffic, one option is to utilize the flexibility of FLASH to selectively designate RAC lines as write-through. Another option might be to periodically flush data back to memory, as is used in paging systems, where modified pages are scheduled for cleaning[SG94].

4.2.6.2 Protocol Processor Speed

We have seen that one of the major factors contributing to reduction in performance is high protocol processor (PP) occupancy. The performance effects of PP occupancy are most apparent in COMA. While some of this overhead is inherent in the complexity of the protocols, we would like to consider the more general effect on performance beyond the FLASH machine, i.e. if the implementations were more efficient, or if they were done in hardware. To approximate such a scenario (and reduce the effect of PP occupancy on the final result) we run the PP at various faster speeds: 2x, 3x, and 6x the current speed. We chose 2x and 3x to study the effect of a node controller of approximately the same speed as the CPU, and 6x to minimize PP occupancy. Because COMA already shows benefits for Raytrace, we focus on Splash, Engineering, and

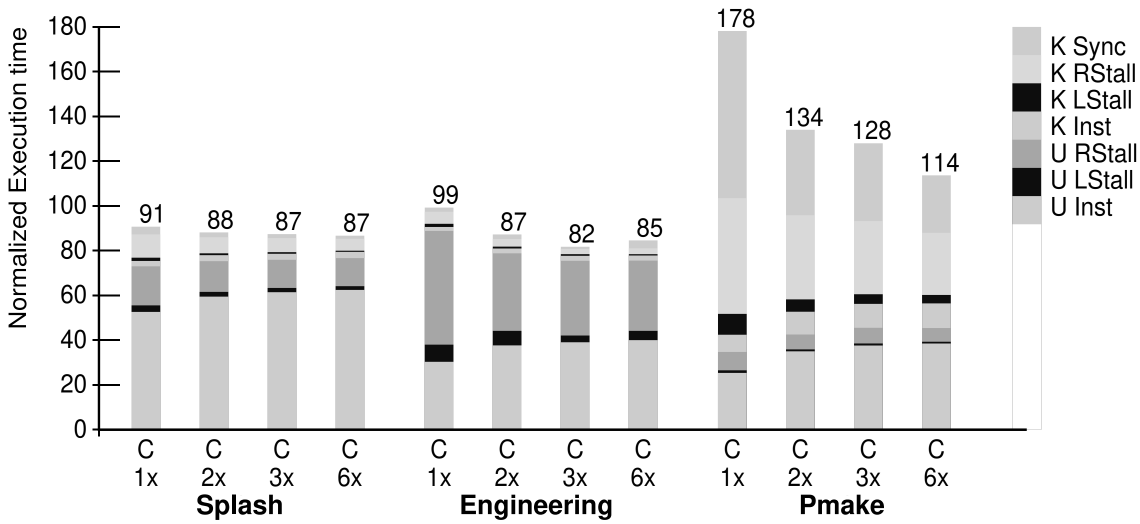


FIGURE 32. Effect of PP speed on COMA (C) performance. All numbers are normalized to the equivalent DynPtr run with the same PP speed. The 1x PP speed corresponds to the default configuration.

Pmake. The results of these runs are shown in Figure 32. In each case the COMA run times are normalized to a DynPtr run with the correspondingly faster PP.

We see that the COMA protocol can benefit from a faster PP. The gain for Pmake and Splash is largest at 6x PP speed, although Splash sees little benefit beyond 3x PP speed. In the Pmake workload the faster PP (lower occupancy) improves performance by reducing both the remote stall time and the synchronization time. However, COMA is still slower than DynPtr. The reason is that COMA is still an intrinsically more complicated protocol than DynPtr. The extra complexity for master tracking and replacement processing with coherence traffic causes extra handlers to run in COMA relative to DynPtr. Even if these handlers are low overhead, more work is nonetheless required for COMA versus DynPtr, and the result is worse performance for COMA.

The other protocols, RAC and MigRep, will see less benefit from a faster PP in most cases because they are not as occupancy-bound as COMA. We will see later on when discussing associativity that for RAC, increasing PP speed still provides the same general performance trends as with the base PP speed.

4.2.6.3 Impact of Associativity in COMA

The preceding sections considered the performance of RAC as the RAC size is reduced, and the performance of COMA as the PP speed is increased. COMA and RAC were both direct-mapped to maintain simplicity, and in the case of RAC, this appears to be a valid design choice. COMA, however, shows a slight degradation in locality from RAC, potentially due to the need to accommodate both local and remote data. Mapping conflicts between local and remote data can cause conflicts that may not occur in RAC. Another important but subtle effect is also important in COMA, and has to do with how reserved memory is tagged.

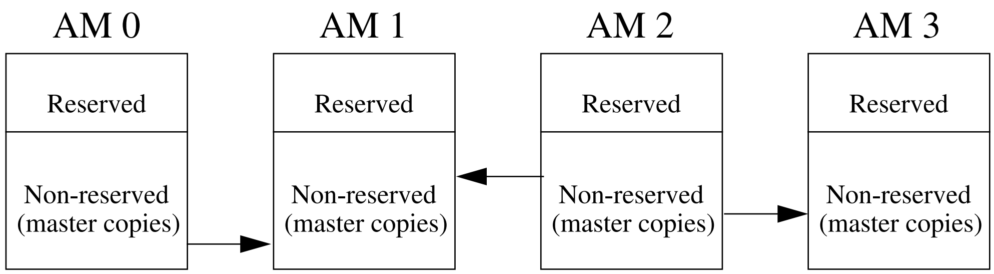


FIGURE 33a. Unstaggered reserved memory on each node. If the OS happens to dynamically allocate reserved memory such that it resides at the same offset on each node, then in a direct-mapped implementation, all replies will displace master copies, and the effect is the same as if there were no reserved memory. AM 1 has requests satisfied by AM 0 and AM 2, and AM 3 has a request satisfied from AM 2, but all replies displace master copies, so only migration is possible, despite the existence of reserved memory**.**

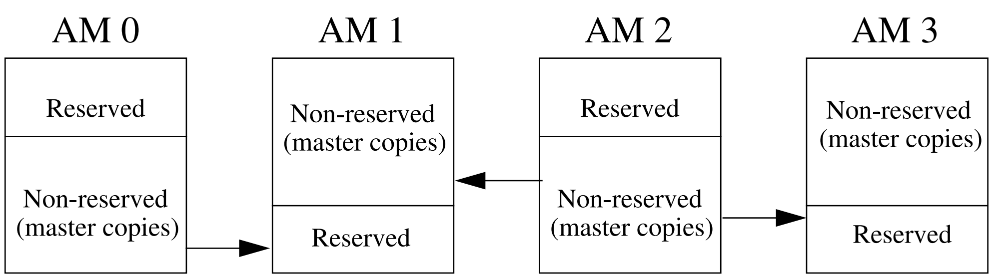


FIGURE 33b. Staggered reserved memory on each node. When reserved memory is staggered on each node, some replies will be placed in reserved memory, not forcing displacement of master copies.

If we were to initially assume that all of memory is reserved (that is, we assume that all of memory is initially empty and therefore devoid of master copies), and only tag a line as not reserved (i.e. tag a line as a either master or a copy) when the OS allocates the page containing that line, we would require some protocol initialization to run whenever pages are allocated (to designate the lines on that page as masters or as a replicas) or deallocated (to invalidate those lines and mark them as empty again). We would also need to modify the OS to make sure it left a certain amount of memory reserved, otherwise we might use up all of memory and only allow migration. Even with these changes, we might encounter pathological cases in which the OS allocates memory in such a way that reserved memory lies at the same offset in memory on each node. In a direct-mapped AM, this would be tantamount to having no reserved memory, since the non-reserved regions on each node would be the same, and any reference to a line would move it from one non-reserved region to another, as shown in Figure 33a.

To avoid the protocol and OS complexities and overheads described above, in our implementation, we statically partition memory into reserved and non-reserved regions, and stagger the reserved regions, as shown in Figure 33b. All non-reserved memory is assumed to contain master copies: any page that is allocated must be chosen from the non-reserved memory, and thus, upon allocation, each line on a page is a master copy.

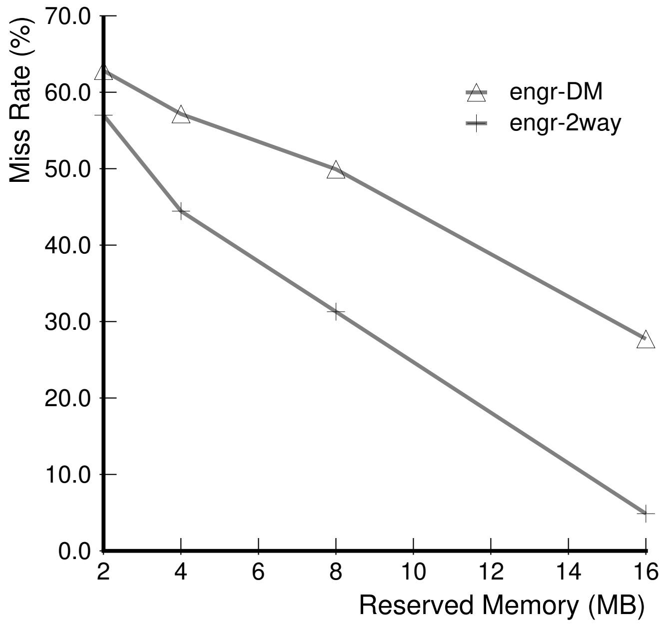


FIGURE 34. Miss rate comparison, two-way set-associative AM vs. direct-mapped AM, varying reserved memory, Engineering.

Any individual memory line can eventually still hold either a master or non-master copy, but the static division provides a simple way to initialize our protocol data structures and avoid runtime overheads of dynamic initialization, like invoking the PP to initialize the directory entries on each page allocation. In the steady state, if all of non-reserved memory were utilized, then all of non-reserved memory would contain master copies, and static initialization scheme would perform as well as a dynamic initialization scheme.

One potential improvement we can consider for COMA is to improve the associativity of the AM. There are two advantages to increased associativity. First, collisions as a result of two addresses sharing the same offset bits can be avoided. Second, if an incoming line collides with a master copy, the other set can be checked to see if it contains a non-master copy, preventing a costly master conflict resolution operation. In effect, incoming data will have a greater probability of encountering a non-master memory location on a Put reply, so master copies will not be evicted and locality may be improved. Implementing associativity in FLASH is somewhat challenging, however, for a variety of reasons. For example, an associative design requires multiple tag checks. In a hardwired COMA design, these checks could be done in parallel, with a multiplexor to choose the appropriate datum. In FLASH, however, no such associative checking facility exists. Instead, each tag check must occur serially in PP microcode. Setting tags and state requires first figuring out which set must be modified, and then performing the modification, using PP microcode. In addition, the PP must issue two memory references and consume two data buffers, even though only one reference at most will be successful. Another difficulty is tracking which AM set an incoming line is meant for. An explicit miss han-

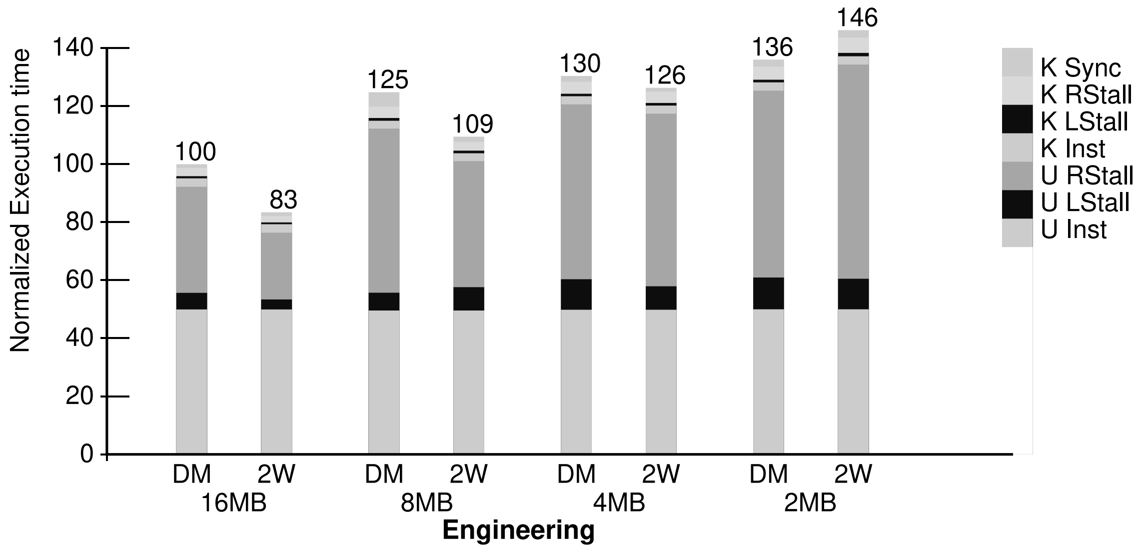
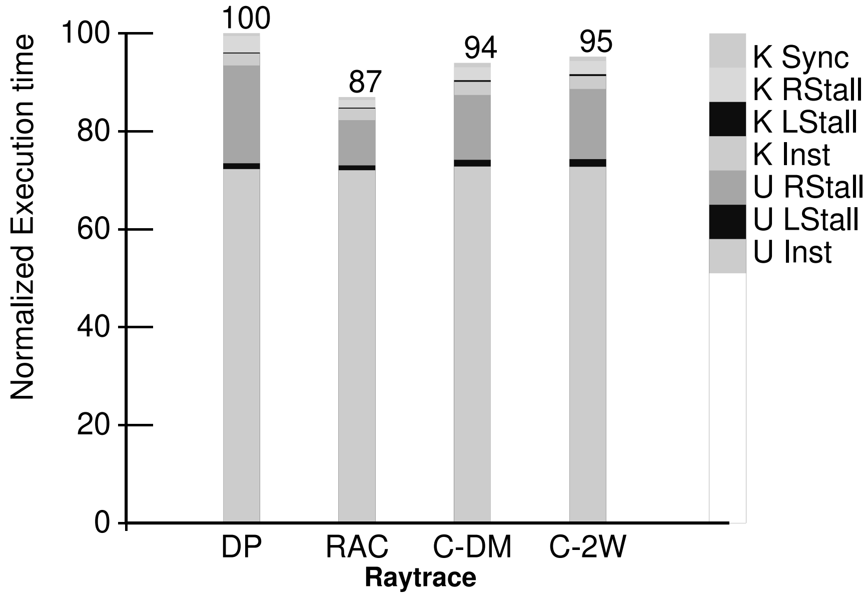


FIGURE 35. Performance of Engineering with 2-way set-associativity and varying reserved memory. The results are normalized to a direct-mapped COMA run with 16MB reserved memory**.**

dling table is used in the processor to track outstanding references and determine which set a reply datum is meant for, but no such table exists in MAGIC for the AM. Instead, a miss handling table must be implemented using PP microcode. One final difficulty rests with determining which set to replace on an AM miss. In a typical processor cache, a random replacement scheme or LRU replacement scheme can be effective. In FLASH, such schemes have to be implemented using PP microcode. To implement priority-based schemes that replace non-master copies preferentially [ZT97] is even slower, because it requires serially checking each line and seeing what state it is in before determining whether to replace it. These schemes have no analogue relative to LRU/random schemes in the processor cache. These processor cache schemes do not care what state a line is in the processor cache, so an implementation in hardware of the AM replacement scheme would not be a straightforward replication of the appropriate processor cache replacement processing hardware.

The overheads alluded to above would dwarf any performance gain that might be seen by increasing associativity in FLASH, so we compare associativity with an accelerated PP (6x faster than the default PP speed), in order to concentrate on the impact of associativity itself. We compare only two-way associativity, because the performance jump from direct-mapped to two-way is typically most significant[PaH96], and because it is far easier to implement two-way set-associativity than it is to implement higher levels of associativity in FLASH.

We first show the change in AM miss rate that accompanies increased associativity. This change in miss rate occurs because of fewer conflict misses between lines at the same offset in the AM, and possibly because fewer master copies are displaced. The miss rates for two-way and direct-mapped AMs for the Engineering workload are shown in Figure 34. Direct-mapped certainly suffers from conflicts, but as the amount of



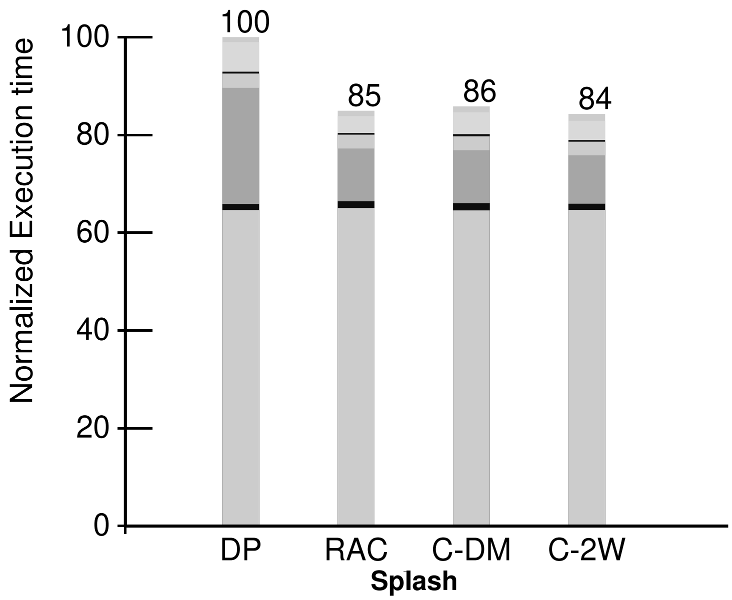


FIGURE 36. Performance of DynPtr (DP), RAC, Direct-mapped COMA (C-DM), and Two-way set-associative COMA (C-2W) with fixed PP occupancy of 0 cycles for Raytrace and Splash.

reserved memory decreases, conflicts with master copies becomes more prevalent. With small enough amounts of reserved memory, no amount of associativity can prevent conflicts with master copies, and the miss rates of direct-mapped and two-way set-associative converge. The results are similar for the other applications, although the miss rates are closer together even at large amounts of reserved memory.

The change in runtime with differing associativity and reserved memory space for Engineering is shown in Figure 35. Despite the difference in AM miss rates, the increased complexity of the two-way design (even with an accelerated PP) causes two-way to be only marginally better than direct-mapped at large amounts of

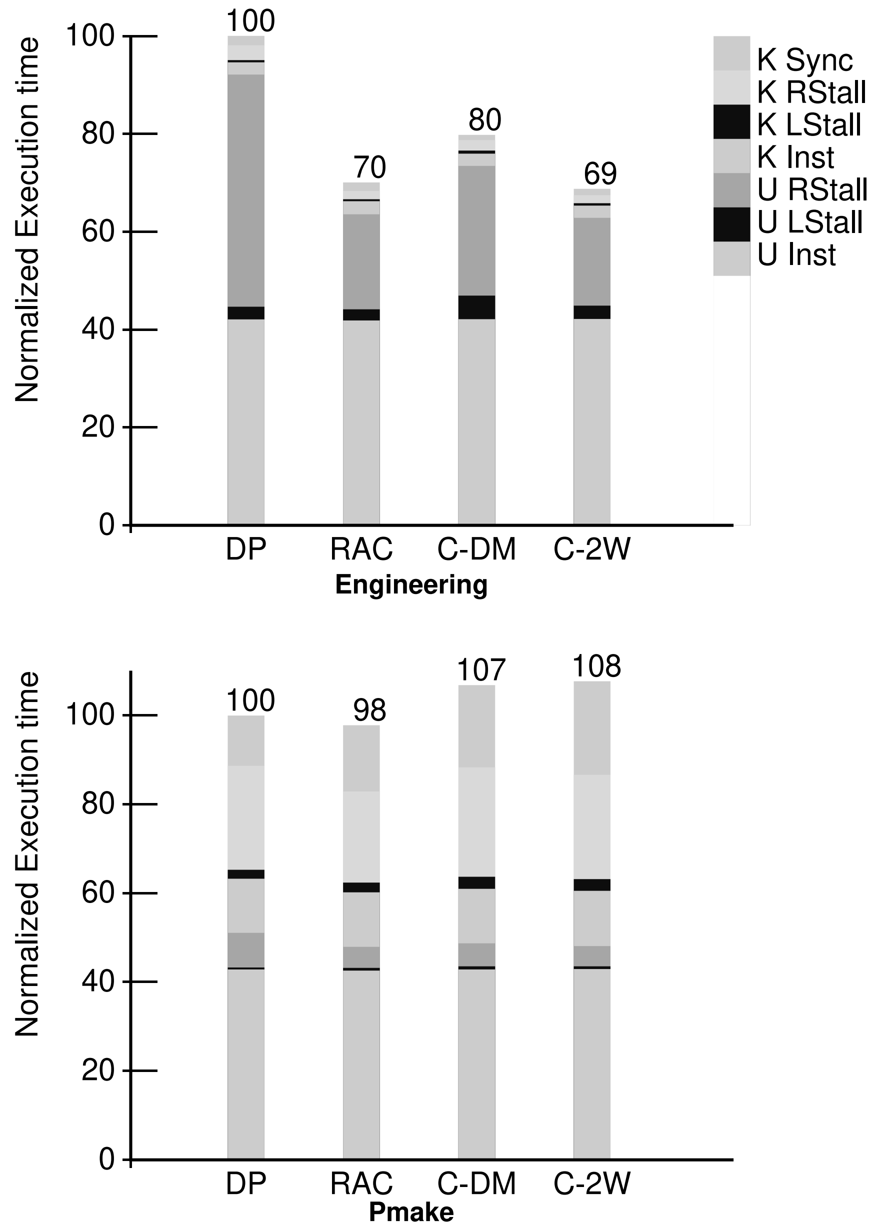


FIGURE 37. Performance of DynPtr (DP), Direct-mapped COMA (C-DM), and Two-way set-associative COMA (C-2W) with fixed occupancy of 0 cycles for Engineering and Pmake.

reserved memory, and worse than direct-mapped at small amounts of reserved memory. Clearly, the change in AM miss rate is small enough that the increased PP complexity required to increase the AM associativity overshadows any gains. The other applications show similar trends.

While accelerating the PP provides a close approximation to a hardwired implementation of each scheme, the base cost of implementing associativity may be overly expensive on FLASH, leading to scaled latencies and occupancies that are still larger relative to hardwired implementations. Latencies and occupancies are typically within 30%, but can be as much as 100% larger for certain handlers, simply because a large number of serial checks of tags and state must be performed, and because of some PP code inefficiencies. Some inefficiencies include saving and restoring stack variables prior to function calls: there is only one load/store allowed per PP cycle, so many of these must be serialized. Some of these problems can be solved through efficient source-code optimizations, and through various tools that have been developed for FLASH[Hei98]. However, the problems described above still exist in some cases.

In order to better estimate the performance of a hardwired implementation, in Figures 36 and 37 we show the runtime of our applications when the PP is assumed to be infinitely fast. That is, memory latencies and queueing at MAGIC interfaces are modeled, but the PP processing time is assumed to be zero cycles. We compare performance of DynPtr to RAC, direct-mapped COMA (DM) and two-way set-associative COMA (2W), using a 16MB RAC and giving 16MB of reserved memory to COMA. As the results show, while the COMA implementations come close to the performance of RAC, and the performance of two-way is even able to exceed the performance of RAC, the differences are so small that there seems to be no compelling reason to implement a complicated scheme like COMA versus a relatively simplistic scheme like RAC. Two-way is in some cases performing slightly worse than direct-mapped, but this is an artifact of variations in OS process scheduling between the workloads.

4.3 Summary of Results

We have seen that approaches that minimize firmware complexity relative to DynPtr, like RAC and MigRep, are able to capitalize on the increased locality that they provide. With sufficient memory for replication, RAC and MigRep provide substantial gains over DynPtr. COMA can provide gains as well, but for miss patterns that stress some of the weaknesses of COMA, like the coherence traffic in Pmake, attempting to push too much functionality into the PP results in worse performance than simpler schemes like RAC and MigRep. Part of this is certainly due to inefficiencies in implementing protocols in firmware, rather than hardwiring them. Because the MAGIC chip services all memory references that miss in the processor cache, it relies on short sequences of instructions to be executed per handler. Simplicity is clearly important, but simplicity is not the only factor, because if it were, then DynPtr would be the most efficient protocol. Instead, the RAC, which strikes a balance between simplicity and locality improvement, emerges as the best-performing protocol in most cases. MigRep is simple, but can miss out on some key locality, while COMA is quite complex, and does not provide sufficient gains to justify its complexity. As we can see, in arbitrarily complex protocols, the occupancies are no longer hidden by memory latencies, and severely impact performance.

Another observation is that the MAGIC ISA is not really a limiting factor in performance. Instead, the limitation is with the amount of parallelism allowed by MAGIC. If operations like tag and state checks could be done in parallel with resource allocation, or if replacement processing could be performed in parallel with other operations, the occupancies in COMA (and to a lesser extent RAC) would be dramatically reduced. To allow such parallelism would involve some degree of speculation, somehow specified by the protocol programmer, since in certain cases the replacement processing or resource allocation is unnecessary. For example, if there were a “replacement thread” that could be invoked, and if it could run at the same time that an incoming handler is being processed, then the efficiency and throughput of MAGIC would be improved. However, the job of the protocol programmer would become tougher, and the MAGIC design would be more complex, making it more difficult to verify and potentially requiring a slower clock rate. The results with a 0-latency PP show the same trends as with the base PP speed, so despite the added complexity with respect to hardwired designs, we can have confidence that the trends we observe are similar to those that would be seen with hardwired designs.

The different regimes in which RAC and MigRep operate well suggest a possible synergy between the two approaches. We explore this relationship in the next chapter.

Chapter 5
MIGRAC: A New Proposal

The results from the previous chapter show that the RAC scheme has many advantages: it can capture coarse and fine grain locality, its eager caching gives benefits from short-term temporal locality, it has relatively low overhead over DynPtr, and it is somewhat robust. However, it also has numerous disadvantages. A statically-sized large RAC will take away memory from the OS and could cause paging. A smaller RAC that does not fit the working set of an application will not provide much benefit and in some cases can degrade performance. Moreover, there is a potential mismatch between the sharing and replication granularity. If a large read-only structure is shared, for example, a number of misses would be required to replicate the whole structure; overhead is paid on each reference, and is not amortized over the entire data structure.

Similar to RAC, MigRep also improves performance (for three out of the four workloads). Its main advantage is that it does not need to explicitly partition memory. The OS is able to dynamically respond to memory pressure and balance replication and paging. By migrating or replicating pages, it is also able to explicitly move the home to local memory, thus removing any future accesses to a remote node. However, MigRep works at the granularity of pages and cannot provide benefit with finer-grain sharing.

There is a natural synergy between these two schemes when combined. The RAC can eagerly capture fine-grain sharing and short-term locality in both the user and kernel references. MigRep can capture long-term sharing of page-sized chunks. MigRep can move such pages to the local node, where they can be accessed more efficiently. Once MigRep migrates or replicates a page to local memory, the RAC no longer needs to cache these pages. This can reduce the capacity misses in the RAC, potentially allowing the use of a smaller statically-partitioned RAC.

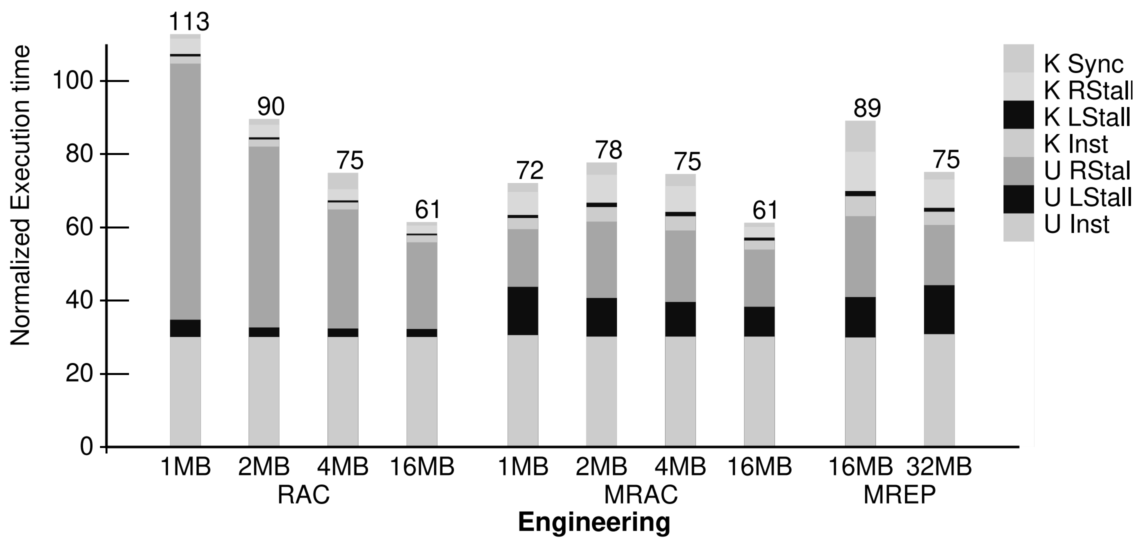


FIGURE 38. Performance of MIGRAC for the Engineering workload. MIGRAC (MRAC) uses 16MB per node as base memory, and adds RACs of size of 1MB, 2MB, 4MB or 16MB. MigRep (MREP) results are shown for both 16MB memory per node and 32MB memory per node. The results are normalized to DynPtr.

Based on this synergy between RAC and MigRep, we propose a hybrid scheme called MIGRAC. In MIGRAC, a portion of memory is statically partitioned to be used as an explicit RAC. The rest of memory is under OS control to be used for backing store, as well as for replication/migration. The MigRep counters and the RAC support code are protocol-level modifications, while the remainder of the MigRep support is in the kernel. Migrations and replications are performed according to RAC misses, not processor cache misses.

MIGRAC uses the RAC to capture fine-grain sharing on a cache-line basis and uses MigRep to exploit coarse-grain sharing on a page basis. Thus, it attempts to exploit multiple grains of sharing. In addition, it attempts to capture different temporal components of locality: short-term sharing is captured in the RAC through its eager caching, and long-term sharing is captured by MigRep through its delayed caching. In addition, because our implementation of MigRep does not support kernel migration/replication (since IRIX5.3 is unmapped), RAC extends the migration and replication capabilities of MigRep through replication/migration of relevant kernel instructions and data.

In order to evaluate MIGRAC, we compare it against a system with only a RAC, and against a system with just MigRep. In Figure 38, we show these results for the Engineering workload, normalized to DynPtr performance. The Engineering workload is particularly interesting because its performance is quite sensitive to the amount of memory available for replication. When the RAC size is varied from 16MB to 1MB, performance relative to DynPtr degrades substantially. When MigRep alone is used with 16MB of memory per node, we see vastly reduced performance compared to MigRep with 32MB of memory per node, because there is insufficient memory to replicate all desired data. In order to stress MIGRAC, we run MIGRAC with

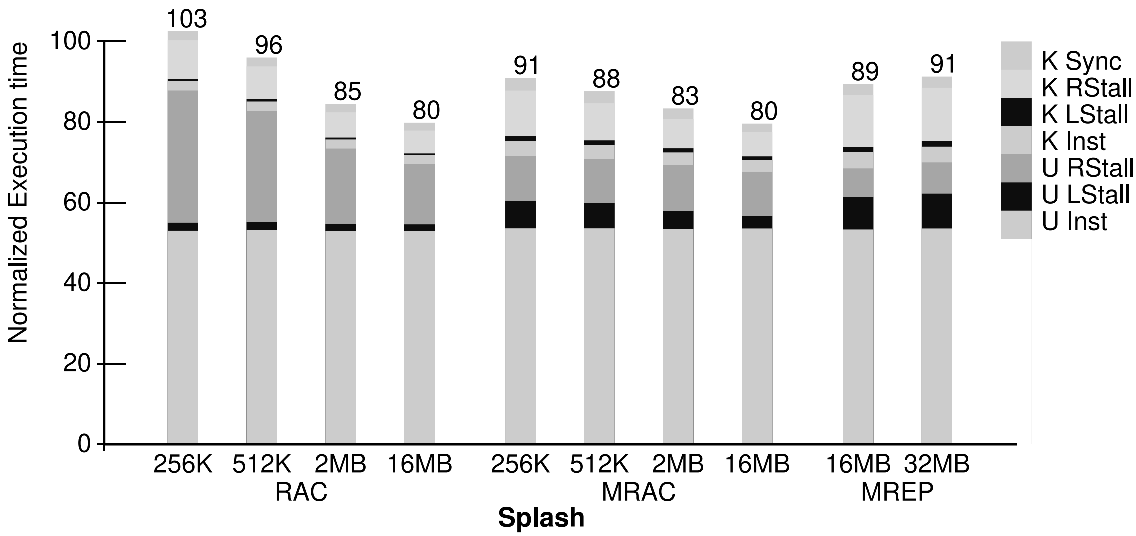


FIGURE 39. Performance of MIGRAC for the Splash workload.

a base of 16MB of memory per node and with RAC memory added to this. For example, in Figure 38, the MIGRAC run at 1MB represents a system with a total of 17MB of memory per node: 16MB “normal” memory, and 1MB of RAC memory. Because the “normal” memory is so small, not all data can be replicated using the MigRep protocol.

As Figure 38 demonstrates, for the Engineering workload MIGRAC is much more robust to changes in memory available for replication than either RAC or MigRep. In addition, consider the 1MB MIGRAC performance. In this case, there is insufficient RAC for the RAC to outperform DynPtr. MigRep does outperform DynPtr, but is impacted by the reduced replication memory available at 16MB per node. The MIGRAC hybrid succeeds in significantly outperforming RAC (by 57%), MigRep (by 24%), and DynPtr (by 39%). Moreover, this performance is almost as good as MigRep with sufficient memory for replication, and is only slightly worse than best-case RAC performance. While RAC sizing is still important, it appears to be less critical in MIGRAC.

The Splash and Raytrace applications perform similarly: the results are shown in Figures 39 and 40, respectively. In each of these workloads, the difference between best-case performance and worst-case performance for RAC and MigRep is much less than in Engineering. Interestingly, with less memory for MigRep, both Splash and Raytrace perform marginally better: this small difference is due to a slightly different initial allocation of data. In each case, MIGRAC begins to outperform RAC and MigRep at a RAC size of around 512KB. MIGRAC is again less sensitive to RAC sizing than a RAC alone, although the base performance of MigRep is good enough with 16MB per node that this robustness is not surprising.

While the previous results demonstrate gains for MIGRAC over RAC and MigRep, and exhibit the synergy between RAC and MigRep, the situation is different for Pmake, shown in Figure 41. In Pmake, MigRep

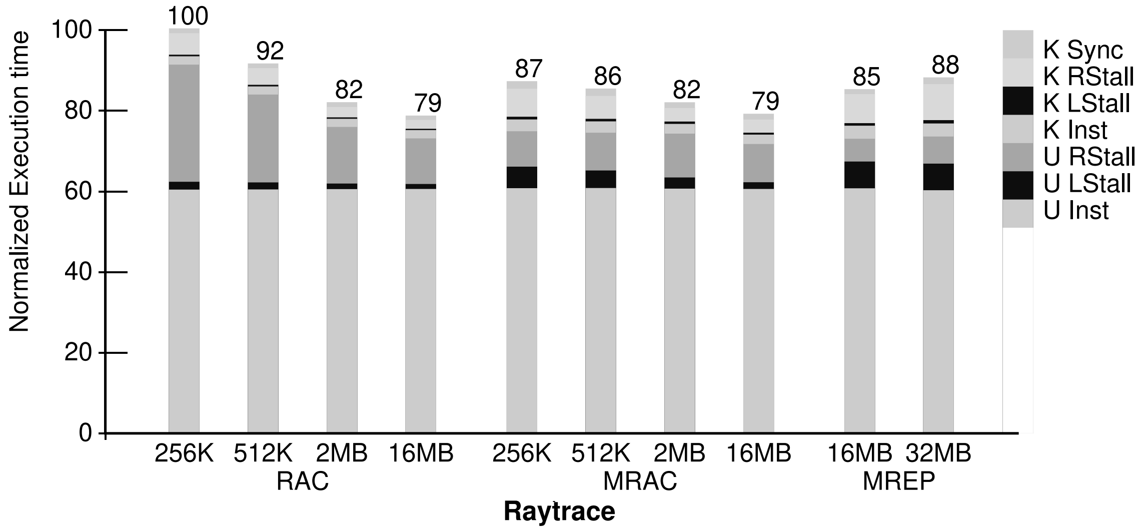


FIGURE 40. Performance of MIGRAC for the Raytrace workload.

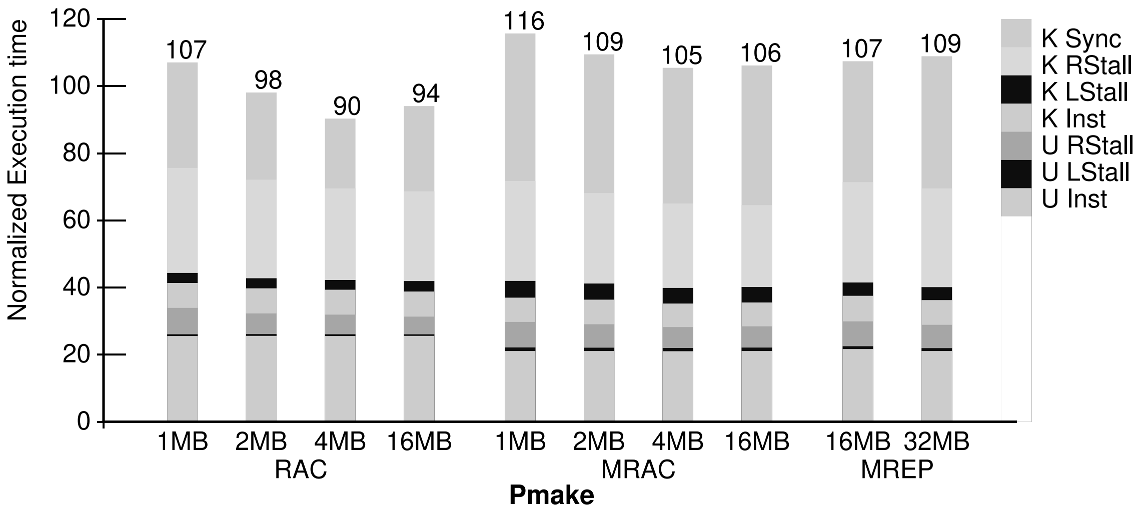


FIGURE 41. Performance of MIGRAC for the Pmake workload.

alone performs worse than DynPtr both with and without memory pressure. RAC alone also performs worse than DynPtr when the RAC size is small. Unlike in the other applications, the MIGRAC hybrid performs worse than both RAC and MigRep when the RAC is too small. We do not see the two approaches complementing each other and improving performance when memory is limited: instead, the relative inability of MIGRAC to deal with coherence misses (i.e., the dominant misses in Pmake), combined with its firmware and OS overhead, causes it to suffer. The degradation of performance even at larger RAC sizes is mostly due to OS overhead with respect to increased locking of page data structures, and the extra checks required to determine if a replication or migration is necessary. One possible way to avoid such overheads might be for

Workload	RAC hit rate (%)		Pages Replicated	
	RAC	MIGRAC	MigRep	MIGRAC
Raytrace	48	56	2661	1083
Splash	40	50	2982	1460
Engr	35	47	3494	2083
Pmake	26	25	664	160

TABLE 16. Robustness of MIGRAC. For Raytrace and Splash, the RAC size used is 512KB. For Engineering, the RAC size used is 1MB. For Pmake, the RAC size used is 2MB. In all cases, the non-RAC memory per node is 16MB**.**

the OS to inform the protocol dynamically when the replication/migration potential is found to be small. For example, the OS might write a variable that the protocol checks periodically to determine if the protocol should continue to count cache misses. If this variable is reset, then the protocol would cease counting cache misses, thereby avoiding the invocation of costly and useless OS policy code.

In three out of the four workloads, MIGRAC outperforms RAC and MigRep. The advantage of MIGRAC stems not just from its performance gains, however: a more subtle advantage of MIGRAC can be seen in Table 16. This table shows that MIGRAC is able to both reduce the number of pages replicated compared with MigRep, and increase the RAC hit rate compared to the base RAC. In 3 of the 4 workloads, half as many pages are replicated in MIGRAC as in MigRep, and in Pmake, MIGRAC replicates only one-fourth the number of pages as is replicated by MigRep. The hit rate gains are modest but noticeable: the RAC by itself has hit rates around 40% (except for Pmake), while the RAC in MIGRAC has a hit rate of around 50% (except for Pmake). Replicating fewer pages implies that there is more free memory for other uses, and having a higher hit rate means that the RAC is better able to accommodate the workload. These numbers clearly point to the synergy between RAC and MigRep that is achieved in MIGRAC.

5.1 Varying Trigger Threshold

The base results assume various RAC sizes and extend them with MigRep, and show that MIGRAC is generally robust across the different workloads. In these results, MIGRAC keeps the same trigger threshold (used for determining when a page is hot) as MigRep by itself. However, because MIGRAC makes policy decisions based on RAC misses, rather than processor cache misses, as is done in MigRep, this threshold may need to be changed to provide better performance for MIGRAC. Using the RAC to monitor the miss pattern to a page results in fewer migrations and replications, since there are typically fewer RAC misses to a line than processor cache misses. While replicating fewer pages is desirable, it is possible that modifying the threshold may provide even better performance. It is thus valuable to consider the sensitivity of MIGRAC performance to the trigger threshold–that is, the number of misses that occur to a page before it is

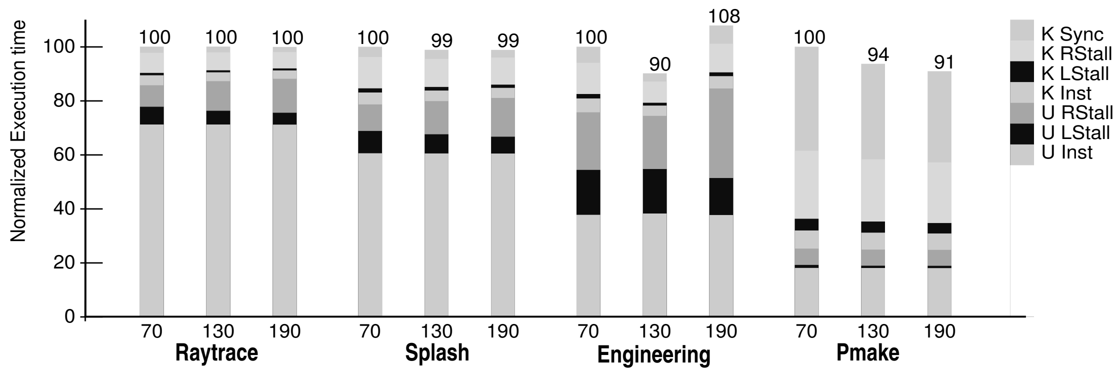


FIGURE 42. Sensitivity of MIGRAC performance to trigger threshold. The trigger is varied from 70 misses to 190 misses, normalized to the runtime at 70 misses. The default trigger for the original runs is 130 misses. For Raytrace and Splash, the RAC size used is 512KB. For Engineering, the RAC size used is 1MB. For Pmake, the RAC size used is 2MB. In all cases, the non-RAC memory per node is 16MB**.**

Workload	Number of Pages Migrated			Number of Pages Replicated		
	threshold 70	threshold 130	threshold 190	threshold 70	threshold 130	threshold 190
Raytrace	1436	833	583	2245	1083	721
Splash	5769	3625	2820	2897	1460	1067
Engr	7094	6312	4483	4009	2083	1541
Pmake	2067	177	23	710	160	68

TABLE 17. Number of pages migrated/replicated as trigger threshold is varied. The default trigger threshold is 130**.**

considered hot and a replication/migration decision is performed. The interplay between the threshold and replication is complex with the addition of the RAC: the RAC may succeed in filtering references such that no replication is needed (since the RAC replicates the data adequately), or may simply make it more difficult to replicate pages that should be replicated. In general, varying the trigger threshold provides a trade-off between increased memory locality resulting in reduced stall time and the increased kernel overhead to migrate and replicate pages[VDG+96]. Clearly the best operating point is highly dependent on miss latencies, kernel overheads, and the amount of time spent on memory stalls. These factors are highly workload-dependent: a workload with a large number of processes may, for example, incur more kernel overhead for load balancing, and the resulting memory stalls may be higher because a process is rescheduled on a node that is far away from the data that the process accesses.

Figure 42 shows the execution times of MIGRAC with varying trigger threshold. As the figure shows, there is no clear trend with varying the trigger threshold. As seen in [Ver98], trigger thresholds based on processor cache misses are very difficult to choose. The addition of a RAC makes the thresholds even more difficult to

Workload	% Misses Satisfied in Local Memory (not including RAC hits)			RAC hit rate (%)		
	threshold 70	threshold 130	threshold 190	threshold 70	threshold 130	threshold 190
Raytrace	51	41	37	56	56	54
Splash	43	43	42	51	50	50
Engr	29	28	31	49	47	45
Pmake	27	31	25	25	25	26

TABLE 18. Locality changes at different trigger thresholds.

choose. For example, in Engineering, lowering or raising the trigger threshold results in worse performance than our default value of 130. Table 17 lists the number of pages migrated and replicated at each of the thresholds, and Table 18 shows the corresponding changes in references satisfied in local memory due to page migration/replication (i.e. excluding RAC hits) and the changes in RAC hit rate. From Table 17, we see that the number of replications in Engineering at trigger 190 is close to the number at 130, which means that the delayed replication of pages hurts performance (as seen in the increased user remote stall time in Figure 42). The number of replications at trigger 70 is almost twice that at trigger 130, resulting in larger kernel overheads but small increase in locality. In Pmake, increasing the threshold provides gains because it reduces the overall kernel overhead by having fewer pages replicated and migrated. Interestingly, the number of misses satisfied in local memory increases as we change from a threshold of 70 to a threshold of 130, although the RAC hit rate remains the same. The subtle interactions between RAC and MigRep make it difficult to predict such locality benefits and their corresponding changes in performance. For Splash and Raytrace, varying the threshold shows no effect in terms of the RAC hit rate. For Raytrace, the misses satisfied in local memory are dramatically reduced from threshold of 70 to threshold 190, but again, the interaction between increased kernel overhead and increased locality yield no performance effect. For Splash, the impact of changing the threshold on the number of misses satisfied locally is minimal. There are much fewer migrations/replications when going from threshold 70 to threshold 130, and less of a difference between threshold 130 and threshold 190. It seems that many of the migrations and replications are unnecessary, but the resulting overheads impact performance only slightly.

5.2 Summary of Results

The MIGRAC scheme exploits the synergy between RAC and MigRep. In compute-server workloads, we see that the demands of the system can change frequently, and statically-sized structures, as in RAC, can perform well in some cases and perform poorly in others, with quite a large variance between the approaches. The MigRep scheme is similarly sensitive to the amount of memory on each node that is available for replication. We have even seen cases in which limiting the amount of memory on each node provides *better* performance for MigRep than giving essentially infinite memory. Our MIGRAC design with limited memory for MigRep and limited RAC usually performs better than a memory-limited MigRep or memory-limited RAC individually. The MIGRAC scheme is also more robust across wider ranges of RAC sizes, removing some problems associated with static sizing of the RAC. Although applications like Pmake that are dominated by coherence misses can perform worse with MIGRAC than with RAC or MigRep alone, the performance degradation is relatively small, and the gains for other applications outweigh this performance impact. The key is allowing the RAC to store only what it needs to store, and allowing coarsely-shared data to be moved by the OS. This sharing at multiple granularities has clear benefits in other applications not studied here: for example, in a database, it may be the case that certain pages are read-only, and others are read-only except for some locks that reside on that page. The read-only pages can clearly benefit from a MigRep-style replication strategy of MIGRAC, while the other pages can replicate the necessary data using the RAC of MIGRAC.

Chapter 6
Related Work

In this thesis we have explored some of the options available to system designers to transparently decrease the fraction of data misses serviced remotely in DSM systems. The schemes we have studied are (i) base CC-NUMA, (ii) CC-NUMA+RAC, (iii) COMA, and (iv) CC-NUMA+MigRep. Of course, there are a number of other mechanisms for improving data locality. While there has been a significant body of work on data-locality optimizations in multiprocessors, a complete discussion of such work is beyond the scope of this thesis. Instead, this chapter will focus on work comparing hardware and software locality-enhancement schemes to CC-NUMA, and also contrast MIGRAC with other hybrid hardware/software approaches.

6.1 Comparison of Hardware-Based Approaches

Singh *et al*[SJG93+] compared the performance of CC-NUMA and COMA machines, studying performance of scientific applications on actual implementations: the CC-NUMA machine was the Stanford DASH multiprocessor[LLJ+92], and the COMA design the KSR-1 hierarchical COMA design [FBR93] built by Kendall Square Research. Their study shows that for scalable applications, the benefits of COMA come primarily from migration, rather than replication: specifically, in applications that do not perform data placement, COMA can migrate data where it is used. However, poor data placement is not a sufficient condition for performance gains relative to CC-NUMA, since the direct overhead of a COMA approach can offset any such gains. Our results show a similar offsetting of gains due to COMA overhead. Singh *et al* were unable to compare actual execution times because the underlying architectures were different in terms of clock speed, interconnect, and other system assumptions. In contrast, this thesis is able to compare different architectures due to the flexibility of FLASH in allowing each architecture to be implemented on the same base platform.

Stenstrom and Joe [SJG92] compare COMA-F performance versus CC-NUMA performance on SPLASH workloads. They find that COMA-F outperforms CC-NUMA on all the workloads they study. Joe considers other applications in [Joe94] and shows that CC-NUMA can outperform COMA-F when coherence traffic dominates. Both [SJG92] and [Joe94] are not based on real protocol implementations, but instead on high-level models of the protocols. As a result, the estimated costs of various protocol operations and the coherence controller occupancy may be inaccurate, leading to more optimistic results than those presented in this thesis.

Zheng and Torrellas[ZT97] build upon the COMA-F work of Stenstrom and Joe and provide one of the first evaluations of COMA-F versus a RAC design. They find that RAC outperforms COMA in most cases, unless the sharing pattern is migratory. Another important observation they make is that for a RAC that is the same size as the amount of reserved memory in COMA, the RAC is better able to utilize this "overhead" memory. A simple example illustrates this point. Consider a COMA system containing a 2-way set-associative 32MB AM, with 16MB reserved memory. Compare that system to one with 16MB 2-way set-associative RAC and 16MB normal memory. For the COMA design, half of memory is potentially filled with master copies (possibly one per set). As a result, there is only one copy per set on average that is not a master copy. Effectively, the space of reserved slots is direct-mapped, since there is only one reserved slot per way. Contrast this situation with RAC, in which all of the RAC is available for copying, and thus remains 2-way set-associative. While our results corroborate their results, their methodology and workloads are quite different: their evaluation uses scientific workloads and does not use actual implementations of the COMA and RAC schemes.

The previous studies have compared hardware-based CC-NUMA to hardware-based COMA. The SC-COMA[GMJ+96] group at USC extends such a comparison by comparing a software-based COMA implementation to CC-NUMA. Specifically, the protocol engine in SC-COMA is emulated in software executed on the main processor. This protocol engine thread runs in sub-kernel mode, invoking OS trap handlers on remote misses. They find that CC-NUMA outperforms SC-COMA in all cases, by as much as 144% in some cases (e.g. FFT, in which coherence traffic hurts performance), but that in most cases SC-COMA is still competitive with CC-NUMA. They extend this work by comparing SC-COMA to an all-hardware COMA[MGD97]. A hardware COMA is simulated by assigning zero cost to coherence handlers, following the methodology of [HKO+94]. The authors note an 11-56% slowdown of the SC-COMA over a hardware-only COMA protocol. The SC-COMA protocol differs from our work in numerous respects: first, the protocol runs on the main processor, rather than on a co-processor; second, the software is written as OS code, rather than as firmware microcode, and thus is shielded from many of the resource issues faced by our COMA design; third, the evaluation is performed exclusively on Splash applications; finally, the SC-COMA work is intended for use in message-passing, distributed environments, in which network latencies are high enough so that the software overhead time is not the main factor in overall communication time.

The work done by Reinhardt *et al* [RLW94] is similar to ours in that it compares a RAC-like protocol (called Stache) to a hardware cache-coherent machine. The architecture they assume also contains a flexible memory co-processor, rather than a hardwired memory controller. Stache treats a portion of memory as a cache for remote data. Unlike our RAC design, Stache uses a different granularity for communication than for allocation: in Stache, a remote miss causes the page to be allocated locally, but only the requested line is transferred, with fine-grain access control used to identify valid lines in the page. A similar scheme is used in the KSR1[FBR93], and in S-COMA, described below. By allocating at the granularity of pages, the RAC in Stache is fully-associative at the granularity of pages. In our RAC, the page remains mapped remotely, and only the relevant cache line is transferred: the granularity of allocation is a cache line, and the coherence unit is a cache line. While the benefits of an effectively fully-associative RAC are clear, locally allocating pages can lead to internal fragmentation if pages are poorly utilized. In addition, allocating pages locally requires eviction of lines that were previously stored in those Stache pages, potentially causing useful data to be kicked out of the Stache to provide space that is never used. Moreover, continuously allocating local pages can lead to memory pressure.

6.2 OS-Based Schemes vs. Hardware

Most previous OS-based page migration and replication work only considered the benefits of better data locality over straight NUMA[BFS89][CoF89][Hol89][ScD89][BSF+91][LKE91][LHE92] or CC-NUMA[VDG+96], rather than comparing them against other hardware-based replication schemes like COMA or RAC. In NUMA systems, page migration and replication is used as a substitute for hardware cache coherence, rather than as a mechanism for improving data locality over and above straight caching. Adding cache-coherence fundamentally changes the trade-offs involved in replication/migration: with caching, good locality obviates the need for migration/replication. As a result, it becomes critical for the page migration/replication implementation to be low overhead and robust.

Distributed shared virtual memory (DSVM) [Li88] also utilizes page migration and replication as a substitute for hardware cache coherence. Similar to migration and replication in NUMA machines, DSVM synthesizes a shared memory abstraction on top of a distributed system. When a node accesses a page, that page is mapped locally read-only. On writes to that page, a fault occurs and the other copies of the page are invalidated, providing a coherence protocol at the granularity of pages. DVSM suffers from false-sharing problems, and a many approaches have been proposed to alleviate these problems [BCZ90][BZS93][KCZ92]. A number of these schemes rely on programmer annotations to associate synchronization with the variables it protects, thus allowing finer-grain sharing than pages. These schemes have approached the performance of hardware cache-coherent machines, but in workloads with significant communication these approaches suffer.

Stenstrom *et al* [SJG92] briefly consider page migration versus COMA for scientific workloads and conclude that page migration can perform as well as COMA when sharing patterns are coarse. However, this study uses simple estimates for the overhead of migrating or replicating a page. In addition, neither the COMA protocol nor the page migration/replication protocol are complete implementations.

6.3 Hybrid proposals

Our MIGRAC hybrid is most similar to Simple COMA (S-COMA) [HSL94] and R-NUMA[FaW97]. Simple COMA is a hybrid approach which proposes tight coupling between the OS and hardware. Instead of maintaining a separate tag store, the MMU translation hardware is used to determine whether or not a datum is present: on a TLB miss, the line is assumed to be not present. The page is mapped locally in the *page cache* (which is the term for AM in S-COMA), and the only the requested cache line is retrieved from the home (or current owner). If the local page had already had valid data on it, then that valid data must be evicted so that the new data can be copied into that page. Because a page is mapped, and space is allocated for all the lines on that page at allocation time, when a datum is evicted from an S-COMA page, the replacement process is simpler than in COMA-F: the line is simply sent to any AM that has a valid mapping for the page on which that line resides. Having a valid mapping implies that a node can behave as the “home” for a particular line. Of course, either the OS or hardware is responsible for locating such a node, and there are still race conditions in situations in which all nodes try to release a line at once. As a result, replacement processing is still complex. In addition, because every node has its own local mapping for a page, some reverse translation is required when locating a line on another node: the node-local mapping must be converted to a global address when sent to the home. If a node references a datum for the first time, then the virtual address must be translated into the global address before the request is sent to the home[SCW+95]. The home then takes the global address and uses it to determine the local mapping for the page containing that line. This translation at the requestor and at the home adds overhead.

In S-COMA, coherence granularity is kept separate from allocation granularity: coherence is kept on cache lines, with fine-grain access state associated with each line on a page. Many mappings of a page may exist, but with the use of the fine-grain access state, the underlying cache-coherence protocol is responsible for keeping coherence between any replica lines. The Stache protocol described above is in fact a variant of Simple COMA. [SWC+95] compare Simple COMA to CC-NUMA and COMA-F using scientific applications, and conclude that Simple COMA and COMA outperform CC-NUMA when memory utilization is low, but perform worse than CC-NUMA under memory pressure. However, this evaluation does not involve a real implementation of Simple COMA. Evidence from other sources [BaT98][Kur98] suggests that S-COMA can perform far worse than COMA-F, and can suffer relative to CC-NUMA at moderate memory pressure. The WildFire prototype designed by Hagersten and Koster [HaK99] implements a variant of S-COMA similar to R-NUMA (described below) in which there is a static home for each page, and show throughput gains against base cache-coherence on an OTLP benchmark. However, they do not compare their hybrid against a hardware-only RAC.

To cope with some of the drawbacks of pure S-COMA, Falsafi and Wood [FaW97] propose Reactive NUMA (R-NUMA), which merges features of S-COMA and CC-NUMA. R-NUMA dynamically changes the coherence protocol on a per-page basis, switching between S-COMA and CC-NUMA depending on the sharing pattern. R-NUMA relies on a flexible controller like MAGIC in order to switch rapidly between protocols. On evaluations of scientific applications, R-NUMA never performs worse than both CC-NUMA and S-COMA, and often is better than both. The R-NUMA evaluation was not done using actual OS and coherence protocol implementations, so it is not clear whether the overheads assumed are realistic.

MIGRAC differs from S-COMA and R-NUMA in a number of ways. First, MIGRAC performs both migration and replication of pages, not just replication, as in S-COMA and R-NUMA. S-COMA and R-NUMA do not perform replication in the same manner as MIGRAC: R-NUMA and S-COMA simply allocate local copies, filling them on demand a cache line at a time, as opposed to MIGRAC, which creates a local mapping and moves all the data on the page at mapping time. Both S-COMA and R-NUMA can induce memory pressure by replicating too many pages, and both perform eager replication. In addition, if only a few lines on each page are used, then memory fragmentation results. In contrast, MIGRAC can perform migration, which keeps the amount of memory used in the system constant, and can dynamically respond to memory pressure by choosing not to replicate or migrate pages. MIGRAC can thus utilize memory more efficiently by not creating unneeded replicas, and also prevents unneeded replicas through its delayed replication. Unlike S-COMA, MIGRAC waits for a consistent miss pattern to emerge on a page before migrating or replicating that page. Unlike R-NUMA, MIGRAC uses one hardware coherence protocol for all pages: replicated pages are kept coherent by the operating system and require no reprogramming of MAGIC handlers. Using multiple coherence protocols on a per-page basis requires the overhead of switching protocols (for example, reprogramming the hardware JumpTable) and can hurt performance of the protocol processor instruction cache.

There has been a wide body of work on other S-COMA hybrid schemes. AS-COMA [KCK+98] attempts to solve the excessive replication problems of S-COMA by throttling replication at high memory pressure. At low memory pressure, pages are replicated as in S-COMA, while at high memory pressure, pages are mapped as CC-NUMA and not replicated. Kernel effects are included in this study, which was performed on scientific applications. AS-COMA outperforms R-NUMA at all memory pressures due to its mechanisms for avoiding thrashing, and suffers only marginal losses compared to CC-NUMA. One drawback of AS-COMA, however, is its eager replication: it is conceivable that a number of initial misses may artificially fill the page cache, requiring the remainder of the pages to be allocated in CC-NUMA mode, even if they would benefit from page replication.

The PRISM project [ELP+98] also attempts to alleviate the replication issues of base S-COMA through a throttling approach similar to AS-COMA's. In PRISM, a page can be mapped as S-COMA, or can be mapped in CC-NUMA mode (called LA-NUMA, since it is locally addressable, but has no local storage). Each node can have the same virtual page mapped in a different mode, so someone can have an S-COMA version while someone else maps it to LA-NUMA. A coherence controller performs the appropriate forwarding of requests based on the physical address (whether the physical address has real storage on the node, or is "imaginary" in the case of LA-NUMA). A number of policies for replication are evaluated in PRISM. One policy allocates pages in S-COMA mode until 70% memory pressure is reached, and then allocates pages only as LA-NUMA. Another policy allocates pages in S-COMA mode until 70% memory pressure is reached, and then evicts pages from the page cache (remapping them as LA-NUMA) according to the utilization of the cache lines on the page (i.e. if a page is heavily utilized, it remains in the page cache). A final policy variant allocates in S-COMA until 70% memory pressure, and then evicts pages using an LRU policy. Each of these policies outperforms S-COMA with 70% memory pressure, except when the application exhibits severe conflict misses.

Similar to MIGRAC, PRISM also supports page migration. However, PRISM's page migration differs from that used in MIGRAC. In PRISM, lazy page migration is employed: there is a notion of a static home, which always keeps the identity of the current home. Each node contains a guess as to the current home of the page. When the guess is wrong, the static home is consulted. When a node wishes to move a page, it coordinates with the home to move the data and directory information. This scheme is similar to that used in [CGS+94], though in [CGS+94] the granularity of sharing was a user-defined object, rather than a page. Unlike MIGRAC, the coherence controller in PRISM performs different actions depending on the page mode. As a result, it requires a more complex protocol implementation than a simple CC-NUMA or S-COMA design. Moreover, PRISM separates memory that is used for the page cache from that used for normal memory, while MIGRAC can dynamically apportion memory to be used for page migration/replication.

In contrast to the schemes that address excessive memory replication, [BaT98] attempt to solve the problem with page fragmentation in S-COMA by space-multiplexing data from multiple pages into a single S-COMA replica page: cache lines from a number of different pages are replicated into the same S-COMA page, with the page table tracking which virtual addresses have cache lines on the same S-COMA pages. This approach requires OS support to determine sparsely populated pages that are candidates for multiplexing. Hardware support is also needed to determine when cache lines from different virtual pages collide in the page cache. Multiplexed S-COMA is shown to approach the performance of COMA-F in many cases, although COMA-F generally performs better.

A final interesting variant of S-COMA is VC-NUMA[MD98]. VC-NUMA adds a network cache (similar to a RAC) to a design with a page cache. Rather than acting like a conventional RAC, the network cache in VC-NUMA behaves as a victim cache backing up the processor. The VC-NUMA design also includes hardware to prevent thrashing in the page cache, as well as an R-NUMA algorithm for determining when to replicate a page. VC-NUMA performs as well or better than a conventional RAC in many cases. While it performs as well or better than CC-NUMA, and outperforms R-NUMA in all cases, it requires modifications to the bus protocol in order to support victimization. For example, typically when a clean block is evicted, a replacement hint (which contains no data) is sent on the bus. However, in order to capture the data in the victim cache, the data must also be placed on the bus, potentially wasting valuable bus bandwidth. The main advantage of VC-NUMA over MIGRAC is that a datum that is replicated in a given node's processor cache is not also replicated in that node's local memory, so there are no redundant copies of data on a node. However, because the RAC in MIGRAC does not enforce inclusion between the processor cache and the RAC, MIGRAC can provide similar memory savings and also show similar performance. Because MIGRAC can be implemented with no changes to the processor's replacement protocol, it is more practical in a design like FLASH that uses a commodity processor.

A number of the schemes above, including MIGRAC, attempt to exploit granularity at multiple grain sizes. MIGRAC attempts to exploit cache-line sharing using the RAC, and coarser sharing using page migration/replication. Other approaches, including MGS[YKA96] and SoftFLASH[ENC+96] utilize small multiprocessors that are connected by a scalable network. In MGS, the small multiprocessors are CC-NUMA multiprocessors, while in SoftFLASH the small multiprocessors are SMPs. Within the multiprocessor building blocks (hereafter referred to as nodes), hardware cache-coherence is used, and between the nodes a DSVM protocol is used. To compensate for the large grain of sharing between nodes, various techniques for allowing multiple writers to a page are employed. These schemes do not use hardware cache-coherence between nodes, as is done in MIGRAC. While MIGRAC has only been evaluated in the context of a low-latency, high-bandwidth interconnect, it is expected that as the latency to remote memory increases, as in these MGS/SoftFLASH systems, MIGRAC will show even more benefits than it shows now because of locality enhancement, especially because the impact of protocol processing time in the OS or MAGIC will be reduced relative to network overhead.

6.4 Summary

Early shared-memory designs relied mostly on replication of cache lines. This replication occurred in processor caches in CC-NUMA designs and in memory in COMA designs. The other end of the spectrum involved software-only approaches, like OS-assisted page migration and replication. Recent work has shown that considering hardware-only or software-only approaches does not provide the best gains, because both approaches miss important overlapping operating points in which there is a combination of coarse- and fine-grain sharing. In particular, while hardware typically performs well at replicating small chunks of data, software replicates/migrates most effectively with large chunks of data. The hybrid systems that have been developed that utilize both coarse- and fine-grain sharing have thus emerged as viable options for best performance. Much of the hybrid work today focuses on allowing multiple coherence protocols to co-exist even at the hardware level, with some operating system support for some sort of page-level sharing. Systems like R-NUMA, MIGRAC, PRISM, and VC-NUMA that allow both CC-NUMA and COMA coherence protocols on the same machine, so-called *Unified Architectures*[Bay99], are becoming more common for providing robust performance.

Chapter 7
Concluding Remarks

Our work has been motivated by several considerations. First, despite the wide range of replication and migration schemes proposed in the literature, there is little data comparing these approaches on the same platform. As a result, it is difficult to determine which is the best approach from the system designer's perspective, or determine if there is a better scheme than those previously developed. Second, a number of the proposed schemes have drawbacks that are not evident without real implementations, and this work attempts to shed light on such drawbacks. Finally, the utility of a single pool of memory and a flexible memory controller in providing varied mechanisms for data migration and replication has never been explored: studying this architectural model provides another viable point in the design space for computer architects.

This thesis is novel in several respects: first, it compares real protocol implementations of both hardware-based and software-based migration/replication schemes versus CC-NUMA. Previous work focussed either exclusively on hardware replication vs. CC-NUMA or software approaches vs. CC-NUMA, or considered high-level models of protocols rather than implementations actual protocols. Second, our work uses compute-server workloads rather than exclusively parallel applications. The use of compute server workloads mirrors one of the dominant use of parallel machines today, and also explores an application space not previously explored with this collection of protocols. For example, running a kernel that can move processes around taxes the locality enhancement mechanisms in ways that scientific applications do not. Finally, the MIGRAC hybrid is a novel approach that capitalizes on the synergy between RAC-based designs and OS-base page-migration and replication.

We have seen that adding a simple RAC to base CC-NUMA is quite effective in improving performance (e.g. 64% for the Engineering workload) by increasing the percent of references satisfied by local DRAM. Furthermore, in the case of FLASH, only one additional handler is needed on top of CC-NUMA and the changes to other handlers are minor. We find that performance is sensitive to RAC size, but the degradation is typically graceful. The advantage of a RAC in memory is that its size can be increased at relatively low cost (a system reboot) as the problems running on the machine get bigger.

RAC allows caching of remote data in memory. Depending on the initial placement of data, or on evolving reference patterns in applications, however, this can lead to severe thrashing in memory. For example, consider a process migration: all of the pages may have originally resided on the same node as the requesting processor, but the process migration may have caused the requestor to now access only remote data. In these situations, it is beneficial for memory to adapt to the changing access pattern. One approach similar to RAC is COMA: COMA treats all of memory as a cache, and thus in theory can adapt to the changing reference stream by migrating data where it is used. While we see that COMA can reduce the execution time for some of the applications (14% for the Raytrace workload), the additional protocol complexity is substantial (the code size is almost double that of the DynPtr protocol), and the improvements for the workloads studied are less than those for RAC. Furthermore, the performance can be significantly worse if coherence misses dominate (DynPtr is 78% faster than COMA for the Pmake workload). Most of the additional complexity stems from the lack of a static home for data in COMA, requiring complicated algorithms for tracking data. While protocol processor occupancy is one of the causes of poor performance, there are also more intrinsic reasons, like lower message efficiency for writes and replacements relative to DynPtr. The impact of these intrinsic reasons is demonstrated by showing that even with a six-times faster protocol processor, Pmake still performs 14% faster with DynPtr than with COMA.

An alternative to performing replication under hardware control is to allow the operating system to make policy decisions. By using software rather than hardware, more complicated policies can be used that are perhaps respond more suitably to changing workloads. The OS, which acts at the granularity of pages in order to amortize the high cost of interrupts, can decide based on the reference patterns whether to migrate or replicate a page. This OS-based page migration/replication scheme, called MigRep, can delay movement until a consistent miss pattern to a page occurs, thus providing some confidence that the page will be used when it is migrated or replicated. While MigRep is as effective as RAC for the Engineering workload (33% improvement), in general, it does not improve performance as much as RAC for the other workloads. MigRep, however, shares the low hardware/firmware implementation complexity with the RAC implementation, instead shifting the complexity into the kernel. Despite the increased kernel overhead, MigRep is robust when a workload demonstrates access patterns that would not be helped by data migration and replication.

We show that the MigRep and RAC schemes work in synergistic ways in our hybrid scheme, called MIGRAC. The page migration and replication capability reduce the requirement for a large RAC, and the RAC, in turn, reduces the amount of memory used for replication. We show that the MIGRAC scheme performs well and is robust, increasing the hit rates in the RAC for most applications relative to a RAC-only design, and decreasing the number of pages replicated relative to a MigRep-only design. MIGRAC achieves gains by exploiting fine-grain, short-term sharing using the RAC, and coarse-grain, long-term sharing using page migration and replication. MIGRAC outperforms DynPtr by as much as 64%, RAC alone by as much as 57%, and MigRep alone by as much as 24%.

Looking at our data from a different perspective, the results point to the desirability of a FLASH-like approach of using a programmable protocol processor and a unified directory-memory/cache-memory/main-memory. Given that all of these schemes can be implemented on a single machine, one can imagine booting the machine with a specific scheme so as to provide the best performance for a given workload. While the current state-of-the-art for a MAGIC-like design is perhaps too slow for commercial use, or even provides functionality that users are unlikely to utilize, flexibility has tremendous value in a prototyping or research environment. In addition, as the technology becomes more advanced, it may become much more feasible in high-performance, commercial environments.

7.1 Future Work

While we have performed our evaluation using a very detailed simulation of the FLASH machine, there are still a number of questions remaining to be answered. The first question is relevant to all simulation-based studies: do the results apply to larger problem sizes, longer running applications, and more processors? A simulation study is necessarily limited by constraints of simulation time in performing such experiments. Given that the FLASH machine has been built, the next logical step is to run our protocols on the real machine to confirm the validity of the results. Running on a real machine is particularly important because the performance of data-locality optimizations is strongly dependent on the applications used in the evaluation. Running on the real machine will allow longer-running applications with larger data sets to be run in a reasonable amount of time; as a result, the true impact of our optimizations can be seen. In addition, such studies might reveal some interesting data points. For example, in a long-running application the number of process migrations will be larger than in our simulations, thus strenuously taxing all of our mechanisms. We might see better performance for COMA in the presence of a poor initial allocation if such an allocation causes the RACs to overflow on each node. For MigRep, we can better evaluate situations with high memory pressure and gauge the effectiveness of MigRep in keeping overheads down either through migration or throttling replication.

In addition to running the applications in this study on a real machine, it is also important to examine other application domains. For example, the majority of shared-memory multiprocessors today are used for transaction processing (e.g., OLTP) and web servers. Good performance on these applications is critical in today's marketplace. Because we can run them on a real machine, we are not constrained by simulation time and can truly run realistic sizes of these workloads to test our schemes and determine what performs best.

Another important question that remains unanswered is whether COMA-F outperforms Simple COMA in practice. The conventional wisdom is that an approach like COMA-F is difficult to implement because it requires a great deal of specialized hardware, while S-COMA is more amenable to commodity parts because it requires mostly software changes. We chose to implement COMA-F rather than S-COMA primarily because COMA-F requires only protocol modifications, while S-COMA requires protocol and OS modifications. Although MigRep also requires OS modifications, we were able to use a pre-existing implementation and avoid the complexity of starting from scratch. Previous studies have shown that COMA-F generally performs better than S-COMA[Kur98][BaT98], but these studies did not use real implementations of either COMA-F or S-COMA. The flexibility of FLASH enables us to perform a more realistic evaluation. Using virtual machines[BDG+97] can help get rid of some of the complexity of implementing S-COMA, particularly the difficulty of modifying a commercial OS to provide S-COMA functionality. The reduced complexity of virtual machines stems from three factors[Gov99]. First, the virtual machine implementation requires much less code than a commercial OS (50,000 lines for a virtual machine vs. a minimum of 500,000 for a commercial OS[Gov99]), making it easier to understand where changes are necessary. Second, using a virtual machine allows the introduction of data structures specific to the migration and replication of data, rather than requiring use of existing page data structures, which may be ill-suited to the task. Finally, kernel code and data structures are often mapped using large pages, but this might be too coarse to allow efficient replication/migration; the use of virtual machines allows pages to be mapped at a different granularity from the system page size, facilitating migration/replication of such pages[1]. To implement S-COMA using virtual machines, the virtual machine layer would be modified to provide the OS services for remapping pages, and a protocol like DynPtr could be modified to provide the S-COMA protocol functionality. The remaining issue with such a study involves the associativity of COMA-F, because clearly a direct-mapped implementation is susceptible to pathological cases that would severely impact performance.

One final issue that remains unresolved is the relative performance of the various hybrid schemes described in the previous chapter. While we have chosen one hybrid scheme (MIGRAC) for evaluation, the gains of MIGRAC motivate a more complete evaluation of the other hybrid schemes. Part of the difficulty in evaluating such schemes lies in modifying a commercial OS to perform the given optimizations. However, as we describe above with respect to S-COMA, virtual machines can also help in prototyping hybrid OS-based migration/replication schemes. Even beyond the evaluation of this set of hybrids, one can also imagine implementing a scheme that combines a hybrid cache-coherence protocol with some sort of OS-based replication/migration. For example, a protocol with a RAC component might selective designate some RAC lines as update-based: when writes to these lines occur, their RAC entries are updated rather than invalidated. Although the R10000 processor does not allow its cache to receive updates, a RAC managed by

1. Another issue is that if kernel code/data pages are unmapped, as in IRIX5.3, then they cannot be migrated/replicated at all using the kernel.

MAGIC could certainly be controlled as an update-based RAC. Such a scheme might help solve some of the pathological behavior seen by an approach that uses too large a RAC, and therefore might benefit a hybrid like MIGRAC.

References

[ABC+95] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 2-13, June 1995.

[ABL+91] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95-109, October 1991.

[BaT98] S. Basu and J. Torrellas. Enhancing Memory Use in Simple COMA: Multiplexed Simple COMA. In *Proceedings of the Fourth Annual Symposium on High Performance Computer Architecture.* 1998.

[Bay99] S. J. Baylor. Unified Scalable Shared Memory Architectures. In *Computer Architecture News,* 27(1):10-20, March 1999.

[BCZ90] J. K. Bennett, J. B. Carter, W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming*, pages 168-175, March 1990.

[BDG+97] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*. 15(4): 412-447. 1997.

[BFS89] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Management. In *Operating Systems Review*, pages 19 - 31, vol. 23, no. 5, 1989.

[BrA97] T. Brewer and G. Astfalk. The Evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring '97*. pages 81-86, 1997.

[BSF+91] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, and A. Cox. NUMA Policies and Their Relationship to Memory Architecture. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 212-221, April 1991.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 1993 IEEE CompCon Conference*, pages 528-537, February 1993.

[CDV+94] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, 12-24, October 1994.

[CRD+95] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating System Principles.* December 1995.

[CeF78] L. Censier and P. Feautrier. A New Solution to Cache Coherence Problems in Multiprocessor Systems. In *IEEE Transactions on Computer Systems*. C-27(12): 1112-1118. 1978.

[CGS+94] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *Proceedings of the Eighth ACM International Conference on Supercomputing.* July 1994.

[CKA91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 224-234, April 1991.

[CoF89] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32-43, December 1989.

[CSG99] D. Culler, J.P. Singh, and A. Gupta. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, Inc. San Francisco, CA.

[Dat97] Data General Corporation. Aviion AV 20000 Server Technical Overview. Data General White Paper. *http://www.dg.com/about/html/aviion_av20000_server_technical_overview.html*. 1997

[ELP+98] K. Ekanadham, B.-H. Lim, P. Pattnaik, and M. Snir. PRISM: An Integrated Architecture for Scalable Shared Memory. In *Proceedings of the Fourth Annual Symposium on High Performance Computer Architecture.* 1998.

[ENC+96] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems.* pages 210-221. October 1996.

[FaW97] B. Falsafi and D. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th International Symposium on Computer Architecture.* June 1997.

[FBR93] S. Frank, H. Burkhardt III, and Dr. J. Rothnie. The KSR1: Bridging the Gap between Shared Memory and MPPs. In *Proceedings of Compcon '93.*

[Gha95] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. Ph.D. Thesis. Stanford University. December 1995.

[GMJ+96] A. Gefflaut, A. Moga, J. Jeong, and M. Dubois. Design and Evaluation of a Software-Controlled COMA. University of Southern California CENG Technical Report 96-03.

[Gov99] K. Govil. Personal Communication. October 1999.

[HaK99] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture.* 1999.

[Hei93] M. Heinrich. Personal Communication. June 1993.

[Hei98] M. Heinrich. The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. Ph.D. Thesis. Stanford University. October 1998.

[HKO+94] M. Heinrich, J. Kuskin, D.Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th Conference on Architectural Support For Programming Languages and Operating Systems,* pages 274-285, October 1994.

[HLH92] E. Hagersten, A. Landin, and S. Haridi. DDM−A Cache-Only Memory Architecture. In *IEEE Computer*, pages 44-54. September 1992.

[Hol89] M. Holliday. Reference History, Page Size, and Migration Daemons in Local/Remote Architectures. In *Proceedings of the Third Conference on Architectural Support For Programming Languages and Operating Systems,* pages 104-112, April 1989.

[HS+99] M. Heinrich, V. Soundararajan, A. Gupta, and J. Hennessy. A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. *IEEE Transactions on Computers*: *Special Issue on Cache Memory and Related Problems*. 48(2): 205-217. February 1999.

[HSL94] E. Hagersten, A. Saulsbury, and A. Landin. Simple COMA Node Implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences.* January 1994.

[Joe94] T. Joe. COMA-F: A Non-Hierarchical Cache-Only Memory Architecture. Ph.D. Thesis. Stanford University. October 1994.

[JoH94] T. Joe and J. L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st International Symposium on Computer Architecture,* pages 82-93, Chicago, IL. April 1994.

[Jou90] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* pages 364-373, May 1990.

[KCK+98] C.-C. Kuo, J. Carter, R. Kuramkote, and M. Swanson. ASCOMA: An Adaptive Hybrid Shared Memory Architecture. In *Proceedings of the International Conference on Parallel Processing.* 1998.

[KCZ92] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture,* pages 13-21, May 1992.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994.

[Kur98] R. Kuramkote. Personal Communication. October 1998.

[Kus97] J. Kuskin. The FLASH Multiprocessor: Designing a Flexible and Scalable System. Ph.D. Thesis. Stanford University. November 1997.

[LaL97] 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.

[LEK91] R. P. LaRowe Jr., C. S. Ellis, and L. S. Kaplan. The Robustness of NUMA Memory Management. In *Proceedings of the 13th ACM Symposium on Operating System Principles,* pages 137-151, October 1991.

[LHE92] R. P. LaRowe Jr., M. Holliday, and C. S. Ellis. An Analysis of Dynamic Page Placement on a NUMA Multiprocessor. In *Performance Evaluation Review*, pages 23-34. June 1992.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 125-132, August 1988.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, May 1990.

[LLJ+92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92-103, May 1992.

[LoC96] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308-317, May 1996.

[McC84] E. McCreight. The Dragon Computer System: An Early Overview. Technical Report, Xerox Corporation. September 1984.

[MD98] A. Moga and M. Dubois. The Effectiveness of SRAM Network Caches in Clustered DSMs. In *Proceedings of the Fourth Annual Symposium on High Performance Computer Architecture.* 1998.

[MGD97] A. Moga, A. Gefflaut, and M. Dubois. Hardware vs. Software Implementation of COMA. In *Proceedings of the International Conference on Parallel Processing.* August 1997.

[MIP95] MIPS Technologies, Inc. R10000 Microprocessor User's Manual. 2nd edition, page 112. June 1995.

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. The S3.mp Scalable Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, Aug. 1995.

[PaP84] M. Papamarcos and J. Patel. A Low Overhead Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348-354, June 1984.

[PaH96] D. Patterson and J. Hennessy. Computer Architecture A Quantitative Approach (Second Edition). Morgan Kaufmann Publishers, Inc. San Francisco, CA.

[PuA93] S. Pudar and A. Agarwal. Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179-190. May 1993.

[RBD+97] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, January 1997.

[RHW+95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: the SimOS approach. In *IEEE Parallel and Distributed Technology*, Fall 1995.

[RLW94] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325-336, April 1994.

[ScD89] C. Scheurich and M. Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. In *IEEE Transactions on Computers*, pages 1154-1163, August 1989

[SCI93] Scalable Coherent Interface. IEEE Standard 1596-1992. August 1993.

[SG94] A. Silberschatz and P. Galvin. Operating System Concepts. Addison-Wesley Publishing Company. Reading, MA.

[SHV+98] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 342-355, June 1998.

[Sim92] R. Simoni. Cache Coherence Directories for Scalable Multiprocessors. Ph.D. Thesis. Stanford University (CSL-TR-93-556). November 1992.

[SJG92] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80-91, May 1992.

[SJG+93] J.P. Singh, Truman Joe, Anoop Gupta, and John Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. In *Proceedings of Supercomputing '93,* pages 214-225, November 1993.

[Sou98] V. Soundararajan. The FLASH-COMA Protocol. FLASH Internal Specification Document. http://www-flash.stanford.edu/Doc. September 1998.

[SWC+95] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings Symposium on High Performance Computer Architecture,* January 1995.

[SWG92] J.P. Singh, W. Weber, A. Gupta. Splash: Stanford Parallel Applications for Shared Memory. In *Computer Architecture News*, 20(1):5-44, 1992.

[SwS86] P. Sweazey and A.J. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414-423, May 1986.

[TSS88] C. Thacker, L. Stewart, and E. Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. In *IEEE Transactions on Computers* 37(8): 909-920. 1988.

[TuG91] A. Tucker and A. Gupta. Process Control Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles,* pages 26-40, October 1991.

[VaZ91] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared-Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 26-40, October 1991.

[VDG+96] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 279-289, October 1996.

[Ver98] B. Verghese. Resource Management Issues For Shared-Memory Multiprocessors. Ph.D. Thesis. Stanford University (CSL-TR-98-753). March 1998.

[Ver99] B. Verghese. Personal Communication. July 1999.

[Web93] W.-D. Weber. Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors. Ph.D. Thesis. Stanford University (CSL-TR-93-557). January 1993.

[WGH+97] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. The Mercury Interconnect Architecture: A Cost-effective Infrastructure for High-performance Servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 98-107, June 1997.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995.

[YKA96] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings 23rd Annual International Symposium on Computer Architecture*, pages 44-55. May 1996.

[ZT97] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. In *Proceedings of the Third Annual Symposium on High Performance Computer Architecture,* February 1997.