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

Vijayaraghavan Soundararajan[1], Mark Heinrich[1], Ben Verghese[2], Kourosh Gharachorloo[2], Anoop Gupta[1,3], and John Hennessy[1]

[1]Computer Systems Lab
Stanford University
Stanford, CA 94305
http://www-flash.stanford.edu/

[2]Digital Equipment Corporation
Western Research Lab
Palo Alto, CA 94301

[3]Microsoft Corporation
Redmond, WA 98052

Abstract

Given the limitations of bus-based multiprocessors, CC-NUMA is the scalable architecture of choice for shared-memory machines. The most important characteristic of the CC-NUMA architecture is that the latency to access data on a remote node is considerably larger than the latency to access local memory. On such machines, good data locality can reduce memory stall time and is therefore a critical factor in application performance.

In this paper 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. FLASH is unique in that each node has a single pool of DRAM that can be used in a variety of ways by the programmable memory controller. We use the programmability of FLASH to explore different options for cache-coherence and data-locality in compute-server workloads. First, we consider two protocols for providing base cache-coherence, one with centralized directory information (dynamic pointer allocation) and another with distributed directory information (SCI). While several commercial systems are based on SCI, we find that a centralized scheme has superior performance. Next, we consider different hardware and software techniques that use some or all of the local memory in a node to improve data locality. Finally, we propose a hybrid scheme that combines hardware and software techniques. These schemes work on the same base platform with both user and kernel references from the workloads. The paper thus offers a realistic and fair comparison of replication/migration techniques that has not previously been feasible.

1 Introduction

Shared-memory multiprocessors are increasingly used as compute servers. These systems enable efficient usage of computing resources through the aggregation and tight coupling of CPU, memory and I/O. As processors get faster, the shared bus becomes a bandwidth bottleneck in bus-based multiprocessors. CC-NUMA (Cache-Coherent with Non-Uniform Memory Access time) machines remove this architectural limitation and provide a scalable shared-memory architecture. A typical design has a number of nodes, each node consisting of one or more processors and a portion of the machine's global main memory. The nodes are connected using scalable interconnect technology, and cache-coherence is maintained using a directory-based scheme. Examples include commercial machines such as Sequent STiNG[18], HP Exemplar[4], Data General NUMALiiNE[6], and SGI Origin 2000[16], and academic prototypes such as Alewife[1], DASH[17], and FLASH[15].

The most important attribute of the CC-NUMA architecture is that the latency to access data on a remote node is considerably larger than the latency to access local memory. The ratio of remote to local access times can be as small as 2 (or 3) to 1 for the SGI Origin or as high as 8 to 1 for the Sequent STiNG. On such machines, good data locality can reduce memory stall time and is therefore a critical factor in application performance. However, the dynamic nature of compute-server workloads makes it difficult to ensure good data locality. For example, static data-placement schemes do not work because the operating system moves processes between processors to maintain load balance.

In this paper we study the various options available to system designers to transparently increase data locality for applications. The work is done in the context of the FLASH machine at Stanford (Figure 1). FLASH is unique in that each node has a single pool of DRAM that can be used in a variety of ways by the programmable memory controller.

We first compare the base cache-coherence protocol in FLASH (centralized directory information) with the commonly used SCI protocol (distributed directory information). We then study three ways in which data locality can be increased. The configurations we study are listed below:

- **Base CC-NUMA**: This is the base configuration for FLASH. A portion of the DRAM holds the directory that is used to maintain coherence between the processor caches and memory. Both the dynamic pointer allocation (DynPtr) and scalable coherent interface (SCI) protocols provide cache coherence, but have no inherent mechanisms to increase data locality.
- **CC-NUMA+RAC**: As compared to base CC-NUMA, an additional segment of the DRAM is reserved to implement tags and data storage for a remote-access cache (RAC). RAC is thus a main-memory cache and can be made much larger than the L2 cache of the processors.
- **COMA**: We make a more fundamental change to the protocol structure and convert the whole DRAM to a cache (hence the name Cache-Only Memory Architecture). By treating all of memory as a cache, COMA allows both migration of the "home" of data to local DRAM as well as replication of data.

- **CC-NUMA+MigRep**: As an enhancement to CC-NUMA the kernel can perform page-level migration and replication to increase locality. We modify the cache-miss handlers to use a small portion of the DRAM to keep per-page-per-node miss counts that are used by the kernel to make migration and replication decisions.

There are several unique aspects to this study: (i) Since all of the schemes are implemented on the same piece of hardware, and all protocols are complete and working implementations (including full operating system modifications), this study offers the first realistic and fair comparison of these protocols; (ii) The workloads studied include all operating system effects and kernel references (e.g., earlier COMA studies considered user-mode references only); (iii) This wide variety of schemes, especially including kernel-based migration/replication, has never been pulled together and compared before.

There are two ways in which to interpret the results presented in this paper. First, the results can act as a guide to the desirability of implementing each of the individual schemes in the study. Second, the flexibility of FLASH-like machines in implementing various protocols allows for choosing a specific scheme to achieve the best performance for a given workload.

Our comparison of the DynPtr and SCI protocols shows that the centralized directory information in DynPtr leads to a simpler design and yields superior performance. For this reason, the rest of our experiments assume DynPtr as the base CC-NUMA protocol. Our data-locality results show that the simple RAC scheme can be implemented with low additional complexity, 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 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 56% 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. We propose a hybrid scheme called MIGRAC: kernel-based migration/replication handles coarse-grain locality decisions while the RAC protocol exploits fine-grain locality.

The rest of this paper is organized as follows. Section 2 describes the architecture of the FLASH machine. Section 3 presents a detailed description of the various protocols, and provides a qualitative analysis of their effectiveness for different cache-miss types. We describe our experimental environment and workloads in Section 4. Section 5 presents the performance results for each of our schemes. In Section 6, we explore the sensitivity of the different schemes to various parameters such as memory pressure and protocol processing speed. Section 7 proposes and evaluates the hybrid MIGRAC scheme. Finally, we discuss related work and summarize our major results.

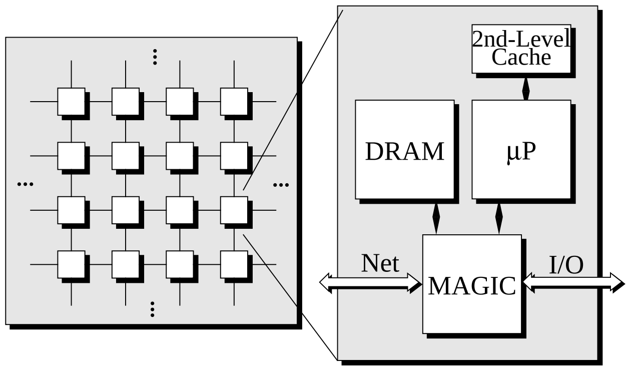

FIGURE 1. The FLASH architecture. The MAGIC chip integrates the CPU, memory, I/O, and network interfaces.

2 The FLASH Machine

The Stanford FLASH multiprocessor is a shared memory machine designed around the MIPS R10000 processor and the MAGIC memory controller chip. The MAGIC chip consists of a dual-issue processor (called the *protocol processor*, or PP), and some specialized hardware to support common coherence actions such as sending messages via the network. The cache coherence protocols are implemented as software routines called *handlers* that run on the embedded protocol processor.

Figure 1 is a diagram of the FLASH architecture, showing the central location of the MAGIC node controller. Figure 2 shows how we partition each node's local DRAM for the five protocols that we study in this paper. The next section provides more details about each protocol.

FLASH provides flexibility in the choice of cache coherence protocol without severely compromising performance. Of all the commercial distributed shared memory (DSM) machines at the time of this writing, only the SGI Origin 2000 has faster remote miss latencies than FLASH, and the local latencies of the two machines are identical. Thus, we present our protocol comparisons in the context of a machine that is achieving good performance, and still show that significant performance improvement over base CC-NUMA is possible by using our locality-enhancing schemes.

3 Protocol Implementations

We begin by describing the two base CC-NUMA protocols along with the three extensions we consider for improving data-locality. We then provide a qualitative discussion on how each scheme responds to different types of cache misses. As we will see in the results section, the centralized DynPtr scheme is superior to the distributed SCI scheme. Therefore, the three locality-enhancing extensions are based on the DynPtr directory data structures.

Before discussing the protocols, we define some terminology that will be helpful for qualitative comparison. Each protocol

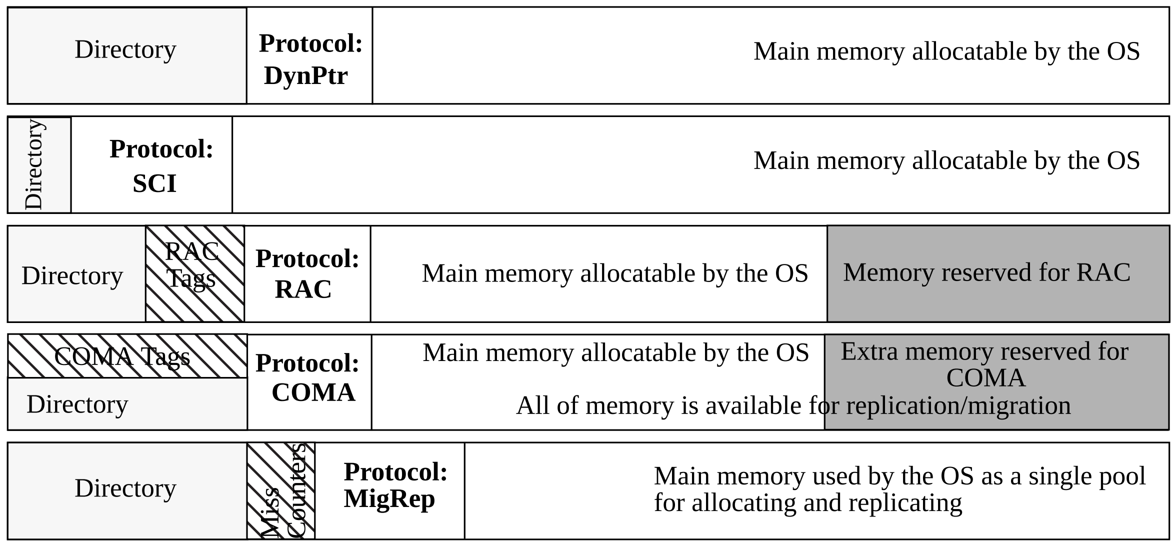
Directory
Protocol: DynPtr
Main memory allocatable by the OS
Directory
Protocol: SCI
Main memory allocatable by the OS
Directory
RAC Tags
Protocol: RAC
Main memory allocatable by the OS
Memory reserved for RAC
COMA Tags
Directory
Protocol: COMA
Main memory allocatable by the OS
Extra memory reserved for COMA
All of memory is available for replication/migration
Directory
Miss Counters
Protocol: MigRep
Main memory used by the OS as a single pool for allocating and replicating

FIGURE 2. The use of local memory on each node by the different protocols. The memory 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.

differs with respect to the complexity of its handlers, which can lead to increased time spent in the PP for a given handler. The two time metrics we use to characterize complexity are *PP latency* and *PP occupancy.* We define PP latency as the time between the PP's receipt of a request (from the CPU, I/O, or network) and its sending of a reply, while PP occupancy is the total time spent within the PP to complete an operation. Latency and occupancy differ, for example, if a handler must do some directory bookkeeping after generation of a reply. High latency and occupancy typically imply complex protocol transactions, and can negatively impact performance.

Base CC-NUMA: DynPtr

Our base CC-NUMA is the dynamic pointer allocation (DynPtr) directory protocol[15][23]. This is the base cache coherence protocol used for FLASH. This protocol maintains centralized directory information, with the complete information for each line available at its corresponding home. Each main memory line has an associated *directory header.* This header contains status bits and a link to a list of node IDs corresponding to nodes currently caching the line. Because DynPtr is a home-based protocol, requests for remote data are always first forwarded to the home for that data. The home node is responsible for either returning the data to the requestor, if it is clean in its memory, or locating the data in a processor's cache, if it is dirty. Most protocol operations at the home node require access to the directory header. Depending on the state of the line and the type of request, a traversal of the linked list of sharers may also be needed.

DynPtr performs replication only at the level of the processor caches, and at the granularity of a cache line. There is no migration of data. All of local memory (minus that used for protocol code, directory headers and the directory pointer store) is available for allocation by the OS because DynPtr does not cache any user or kernel data in local memory. No intervention by the OS is required in protocol operations.

In terms of implementation, DynPtr is the simplest of the protocols we will describe, and is the baseline against which we compare the other protocols. As shown in Table 1, the protocol code consists of 65 handlers totalling 8411 lines of code.

Base CC-NUMA: SCI

Our other base CC-NUMA protocol is the Scalable Coherent Interface (SCI) protocol, also known as IEEE Standard 1596-1992[22]. The main idea behind SCI is to keep a doubly linked list of sharers which, unlike the dynamic pointer allocation protocol, is distributed across the nodes of the machine. To traverse the sharing list, the protocol must follow the pointer in the directory entry through the network until it arrives at the indicated processor. Each processor must maintain a "duplicate set of tags" data structure that mimics the current state of its processor cache. The duplicate tags structure consists of a backward pointer, the current cache state, and a forward pointer to the next processor in the list. The official SCI specification has this data structure implemented directly in the secondary cache of the main processor, and thus SCI is sometimes referred to as a cache-based protocol. In practice, since the secondary cache is under tight control of the main microprocessor and needs to remain small and fast for uniprocessor nodes, most SCI-based architectures implement this data structure as a duplicate set of cache tags in the main memory system of the multiprocessor.[1]

The distributed nature of the SCI protocol has two main advantages. First, it reduces the size of the directory on each node. In SCI, a directory entry contains only a pointer to the first node in the sharing list, resulting in directory entries that are 1/4 the size of

1. Actually most implementations include this tag functionality as part of a larger node cache similar to a RAC. Because we want to study the issues related to centralized versus distributed directory information in isolation, our SCI implementation does not include a RAC.

those in the DynPtr protocol. Since the duplicate tags structure adds only a small amount of overhead per node, SCI has the least protocol memory overhead of any of the protocols in this study. Second, the distributed nature of the SCI protocol can help reduce hot-spotting in the memory system. In DynPtr, unsuccessful attempts to retrieve a highly contended cache block need to be repeatedly reissued to the same home memory module. In SCI, however, the home node is asked only once, at which point the requesting node is made the head of the distributed sharing list. The home node negatively acknowledges the requesting node, and the requesting node retries by sending all subsequent requests for that block to the old head of the list. Many nodes in turn may be in the same situation. Thus the SCI protocol forms an orderly queue for the contended block, distributing the requests evenly throughout the machine. In high-contention situations, such as access to synchronization variables and particularly at larger machine sizes, this even distribution of requests often results in improved application execution time.

The distributed nature of SCI does come at a cost: the state transitions of the protocol are quite complex. This complexity arises from the non-atomicity of most protocol actions and the fact that the protocol has to keep state at the home node as well as duplicate tag information at the requesting nodes to implement the sharing list. Table 1 lists the number of handlers and lines of code in our SCI implementation. Because it is a distributed directory scheme, SCI has 28% more handlers and many more race conditions than the centralized DynPtr protocol. Furthermore, managing the doubly-linked list (especially on cache replacements) results in more handlers invoked per miss than DynPtr, even in the most common protocol cases. Nevertheless, because it is an IEEE standard and applications can potentially benefit from its distributed nature, various derivatives of the SCI protocol are used in several machines including the Sequent NUMA-Q [18], the Data General NUMALiiNE[6], and the HP Exemplar[4].

CC-NUMA plus Remote Access Cache (RAC)

In the Remote Access Cache (RAC) protocol, while most of local DRAM on each node is treated exactly as in DynPtr or SCI, a portion of DRAM is treated as a cache for remotely-allocated data. Thus, remote data potentially resides not just in the processor caches, but also in this specially-allocated portion of local memory (providing a tertiary cache). The concept of a RAC itself is not new. For example, it was used on the DASH machine, although it was motivated by very different considerations[17].

The RAC is a relatively simple extension to DynPtr. The home in DynPtr already maintains directory entries for every memory line owned by that node. The RAC portion of memory will only store lines that are not owned by that node, so the home need not maintain directory information on them. Instead, these unused directory entries can be used to store tag information for the RAC. A simple direct-mapped RAC can be implemented by storing one tag per memory line.

Remote read requests emitted by the processor are first checked against the contents of the RAC, and if there is a tag match, the request is satisfied without requiring the request to be sent to the home node. Thus, the RAC protocol can reduce miss latency and save network bandwidth and protocol processor occupancy at the home. A miss that might have cost 100 cycles to go to the home and return would only cost the local miss penalty of 19 cycles in RAC. On the other hand, the tag check at the requestor increases the latency of remote requests on a RAC miss relative to DynPtr: in DynPtr, a remote request is automatically forwarded to the home node, consuming 3 PP cycles, while a RAC tag check and miss costs 16 PP cycles. In addition, data replies must update RAC state and write the appropriate data to the RAC, thus increasing miss latency and PP occupancy. A write reply sends dirty data to the cache and modifies the state in the RAC to indicate dirty data is in the processor. Because the RAC can potentially hold dirty copies of data, the protocol generates a writeback message when the data is evicted from the RAC (similar to a processor cache). As indicated in Table 1, the static complexity of the RAC implementation is essentially the same as that of the DynPtr implementation. RAC has 66 handlers and 8951 lines of code as compared to 65 handlers and 8411 lines of code in DynPtr.

TABLE 1. Handler Code characteristics. For each protocol, we list the number of lines of source code and the number of handlers. We omit MigRep because it is identical to DynPtr except for the cache counting code, which adds little overhead to DynPtr.

	DynPtr	SCI	RAC	COMA
# lines	8411	8873	8951	15432
# handlers	65	83	66	87

COMA-F

COMA[14] takes a completely different view of memory from DynPtr. COMA treats all of a node’s memory as a cache, or *attraction memory* (AM). Any data, locally or remotely allocated, can reside in any memory. The main benefits of a COMA come from improving locality by dynamically migrating and replicating data into the local memory of nodes at cache-line granularity. Some extra memory may optionally be reserved for COMA to provide room for replication. While the processor cache is generally too small to take advantage of long-term temporal locality, the local memory is expected to be large enough to capture this. However, these benefits come at the cost of implementation complexity. COMA has increased latency at the requestor on accesses that miss in the AM, as well as extra overhead for state maintenance when data is returned. In addition, because all of memory is a cache, care must be taken on all requests to avoid losing the last copy of a datum.

Our COMA protocol is an implementation of COMA-F[14]. COMA-F involves significant changes to the DynPtr protocol, even though it uses the same underlying directory structure. The complexity is inherent in any real COMA-F protocol, not just our implementation on FLASH. While COMA-F was proposed in [14], our paper reports on the first real implementation of the protocol. Therefore, we will describe our implementation of the protocol in some detail.

COMA uses memory as an extension of the cache hierarchy. Therefore, COMA must store tags for each line in memory. In practice, tags are easily implemented in FLASH. Each node in DynPtr already stores directory information for every line on that node, and in COMA we implement similar directory information to store tags (the directory header structure for DynPtr in FLASH has enough unused bits to accommodate these tags). Assuming a direct-mapped AM, a carefully tuned tag match on a read miss request from the processor can be accomplished with zero overhead relative to a local memory read in DynPtr. We opted for a direct-mapped design, because associativity would require serial checks of tags and increase PP occupancy and handler complexity.

COMA improves performance when remote misses are converted to local hits in the AM. Similar to RAC, a miss that might have cost 100 cycles to go to the home and return would only cost the local miss penalty of 19 cycles in COMA. However, *every* cache miss emitted from the processor (local or remote home) must check the local AM before returning data to the processor or forwarding the request to another node. Therefore, data requests that must be forwarded to another node are slowed considerably. Similar to RAC, a miss in COMA to data not allocated in the local node costs 16 cycles of PP latency before it is forwarded to home, as opposed to only 3 cycles of PP latency for DynPtr.

In COMA there is no static home for data.[2] The protocol is responsible for tracking data and keeping at least one copy of the data in the system. This is accomplished by designating one copy as the *master* copy. If a master copy is replaced from the AM because of capacity or conflict, a new owner must be found for that datum. The protocol generates a replacement request to the static home (i.e. where directory information is maintained). However, since the memory at that node may already be occupied by another master copy, the home is responsible for finding another node to serve as master. This is the most significant cause of complexity in COMA-F. Master replacement handlers account for 23% of the handlers in our COMA-F implementation.

Resource management is by far the most difficult issue facing COMA-F on FLASH. Requests and replacements may not be immediately serviceable by the home because of lack of resources. In this event, the requestor must be able to regenerate the request to avoid deadlock. Careful resource management and dealing with resource corner cases account for approximately 4,000 lines of code in our COMA-F implementation. In a hardwired implementation, extra storage may be allocated specifically for handling replacements, thus avoiding many deadlock conditions. This hardware will need to be relatively complex to accommodate worst case conditions.

In summary, while COMA-F has been proposed in the literature, this paper represents its first real implementation. We find that the static complexity is significantly more than the DynPtr and RAC protocols. The number of handlers increases from ~65 for DynPtr and RAC to 87 for COMA, and the number of lines of code increases from ~8500 to 15,432 (Table 1). The majority of the protocol code (~50% of the static code size) is devoted to handling replacements from the attraction memory.

Page-based Migration/Replication

The final scheme that we consider is OS-based page migration and replication[28]. We call this scheme MigRep. 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, sampling is used—only one out of every 10 misses runs the instrumented handler to update the miss statistics. The rest of the migration and replication logic is executed in software by the OS kernel.

The virtual memory (VM) of the operating system 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. Currently MigRep does not migrate or replicate kernel pages because the IRIX5.3 kernel is not mapped through the TLB. More details about the implementation of the MigRep scheme can be found in [28]. The main sources of kernel overhead in page migration and replication 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. When the protocol decides to move or copy a page, the home of the page (i.e. directory information) and the data become local to the node. MigRep differs from RAC and COMA, which just cache the data in local memory, but do not move the directory information. Caching implies that the data may be dropped because of capacity or conflict reasons, and may have to be fetched again from remote memory in the future. MigRep treats all of main memory equally, and does not need to statically partition it. However, it is important to respond to memory pressure and not cause paging because of excessive replication. This is done at the policy level by reducing replications in the face of memory pressure, and through selective reclamation of replica pages. Thus the MigRep scheme is robust to memory pressure [28].

Qualitative analysis

The alternative protocols to base CC-NUMA attempt to reduce application run time by decreasing the memory stall component of execution time via replication/migration of data to local memory. To better understand the potential benefits from the different techniques, we examine protocol behavior under different kinds of cache misses:

- *Capacity and conflict misses* arise because of the limited size and associativity of the processor caches. All of the schemes can help satisfy these 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

2. While the directory information for a given line is always maintained at the static home, the data memory at that home is not reserved for this line and can be occupied by remote lines that map to the same AM entry. Hence the “home” for data is not static.

TABLE 2. Description of the workloads. All workloads are run on eight processor machines.

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)

counts cache misses and waits for a page to get hot, and does not cache eagerly. MigRep also needs to find a consistent miss pattern across the entire page, not just for a cache line. Between COMA and RAC, the big difference is the following: 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.

- *Coherence misses* arise when data is actively being read and updated by multiple processors. 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 and will perform poorly if coherence misses dominate.

4 Experimental Environment

We now describe the experimental environment for the results that we will present in the next few sections. First, we describe the machine configuration that we use for our experiments. We then describe and characterize the workloads we used.

Machine Assumptions

We model our CC-NUMA machine based on the Stanford FLASH architecture[15]. The FLASH machine is currently in the process of hardware bring-up. As this machine is not yet available for experimentation, we simulate our experiments using SimOS[21]. SimOS is a complete and accurate simulator of the FLASH machine. It 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.

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[3]; 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 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. 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 is 864ns.[4] FlashLite, of course, properly accounts for contention at all its interfaces 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 we modified IRIX5.3 to implement the migration and replication of pages.

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 2. Table 3 shows the breakdown of execution time for the workloads when run with the base DynPtr protocol, and Table 4 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.

Single Parallel Application (Raytrace): This workload consists of Raytrace[24], a single compute-intensive parallel graphics algorithm widely used for rendering images. The processes are locked to individual processors, a common practice for dedicated-use workloads. This workload has very little kernel

3. The SGI IRIX kernel assumes the sequential consistency model.

4. 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.

TABLE 3. Execution time breakdown of the workloads under DynPtr.

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 4. Miss Characterization of the workloads.

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

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 4. Therefore, this workload can potentially benefit from improved data locality. Even though this workload is not load-balanced by the OS, the unstructured accesses to the main data structure prevent effective static partitioning.

Multiprogrammed Scientific Workload (Splash): The second workload consists of parallel invocations of Raytrace and Ocean[24]. The applications enter and leave the system at different times, and a space-partitioning approach, similar to scheduler-activations[2][26], is used for scheduling the jobs. Both Ocean and Raytrace have large datasets that overflow the secondary cache. 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 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[26]. The workload consists of copies of two applications. One is the commercial Verilog simulator VCS, simulating a large VLSI circuit. 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 the gnuchess program with four-way parallelism. The workload is I/O-intensive, with a lot of system activity from many small short-lived processes, such as compilers and linkers. 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. The benefits from data locality are not obvious for this workload. We use this workload to focus on the migration and replication potential in the kernel.

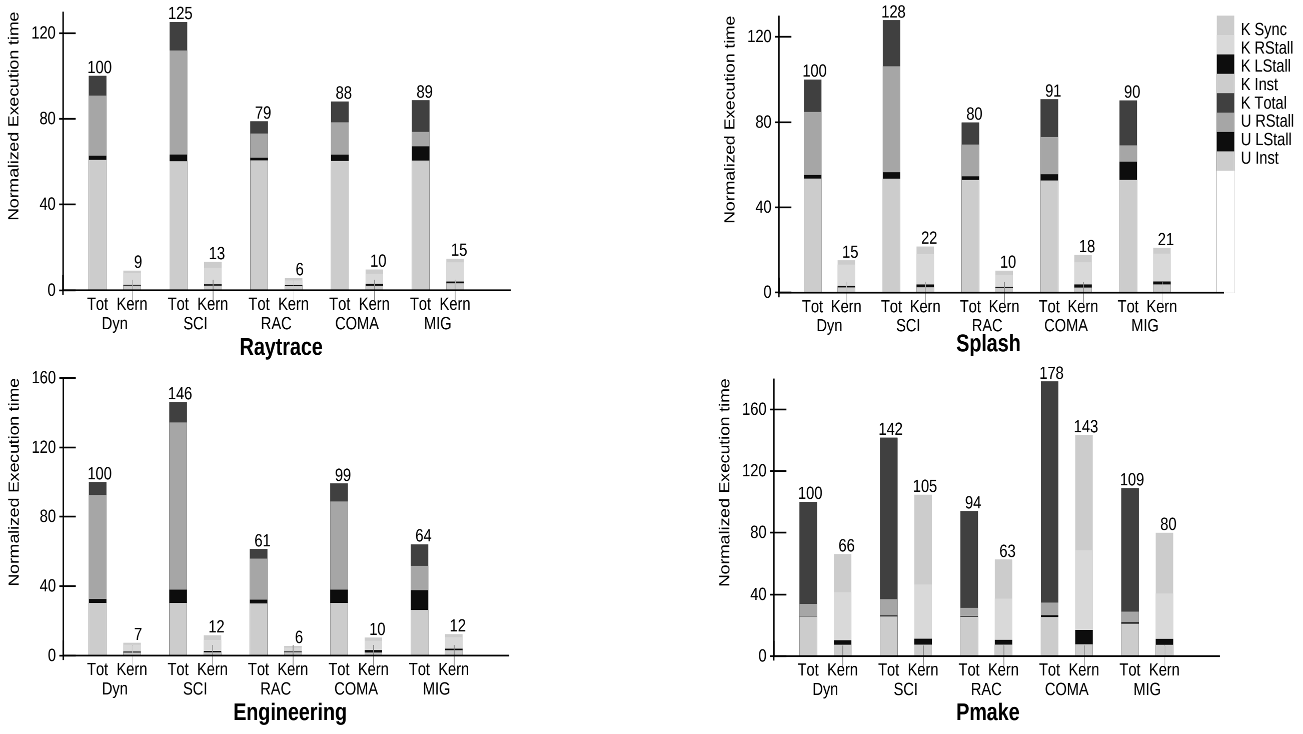

FIGURE 3. Base execution time comparison. For each workload we show the total execution time and a more detailed breakdown of the kernel component of execution time. The execution times are normalized to the DynPtr protocol, except for MigRep. MigRep is normalized to that of the MigRep kernel with migration and replication disabled.

5 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.[5]

For each workload, we first show a graph of the execution time (Figure 3) of SCI and each of the data locality schemes normalized to that of DynPtr. The kernel used for MigRep is different from that used for the other runs[6]. 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. 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 5 provides more detailed data showing the handler costs, PP occupancy, and percentage of misses satisfied from local memory (Local %).

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 and SCI. 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. COMA achieves a lower data locality than RAC, because COMA needs to always find a home for the master copies when replacements occur, even at the cost of throwing away replicas.

5. 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.

6. 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). The characteristics of Pmake, forking and exiting of many short-lived jobs, brings out the worst of this problem.

TABLE 5. Handler and occupancy statistics for the workloads. The average occupancy per handler (occ./hand.) is computed by dividing the total number of occupancy cycles by the number of handler invocations. Local % is the percentage of references satisfied by the local memory.

Protocol	Handler calls/miss	Avg. occ./hand. (PP cycles)	Avg. utilization of PPs(%)	Utilization of busiest PP (%)	Local %
Raytrace					
DynPtr	4.55	15.6	18.2	58.8	14
SCI	6.39	37.1	48.5	64.5	13
RAC	2.18	18.7	13.7	16.2	94
COMA	3.08	25.3	23.1	40.0	81
MigRep	3.08	12.8	13.5	38.1	63
Splash					
DynPtr	4.53	13.8	17.9	38.1	13
SCI	5.74	34.9	45.5	54.2	13
RAC	2.37	20.8	17.1	26.7	90
COMA	3.04	25.8	23.5	45.5	82
MigRep	3.09	13.1	14.4	38.2	62
Engineering					
DynPtr	4.37	13.3	28.1	70.7	10
SCI	5.18	33.1	52.5	73.1	16
RAC	2.04	19.8	33.9	44.7	94
COMA	3.28	28.6	44.9	79.2	71
MigRep	2.80	11.9	24.6	50.7	66
Pmake					
DynPtr	4.69	15.3	14.6	69.5	13
SCI	6.46	33.8	34.7	52.7	13
RAC	3.78	26.8	20.2	58.2	57
COMA	4.98	33.8	22.4	82.7	56
MigRep	4.51	16.3	20.5	80.6	25

- The improved data locality for all schemes results in a smaller number of handlers invoked per miss compared to DynPtr and SCI. Locally-satisfied cache misses do not require remote intervention.
- SCI's distributed nature and management of a doubly-linked sharing list result in a much larger number of handlers invoked per miss than the other protocols. This drives up the average PP utilization and decreases performance.
- The significantly higher handler complexity of SCI and COMA result in a much higher average handler occupancy.
- 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.

The results of the Raytrace and Splash workloads are fairly similar and follow the general trend outlined above. DynPtr is 25% faster[7] than SCI. RAC performs significantly better than DynPtr (27% faster). COMA and MigRep perform about the same, 12-14% faster than DynPtr, but not as well as RAC. Compared to MigRep, COMA has a slightly larger user time and a slightly smaller kernel time for the reasons outlined previously.

The Engineering workload presents a different picture. In this workload the RAC and MigRep schemes both show large gains, 64% and 56% faster than DynPtr, respectively. This is a memory-intensive workload with a high miss rate and consequently a higher average controller utilization. The higher average controller utilization of RAC overshadows some of the gains due to increased locality. As a result, MigRep has lower user time despite significantly lower data locality. MigRep does, however, add kernel time for performing page migration and replication. For the engineering workload, SCI suffers from the fatal combination of having the largest number of handlers invoked per miss and having the largest PP occupancy per handler. The end result is that DynPtr runs 46% faster. For COMA, the occupancy effects for this high miss-rate workload completely nullify any gains from better locality.

The Pmake workload is different from the previous three because it spends a larger fraction of its time in the kernel than in user mode, it consists of many small jobs rather than a few larger ones, and a large fraction of its misses are coherence misses, not capacity. Most of the kernel misses go to node 0, leading to extremely high controller utilization on that node. 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.

Interestingly, SCI has the largest average PP utilization, but the smallest maximum utilization on any one node. Again, the Pmake workload displays significant hot-spotting behavior, and SCI can improve performance in such cases by more evenly distributing the requests throughout the machine. In this case, the hot-spotting is not severe enough for SCI to overcome its higher overhead and DynPtr is still 42% faster. SCI is, however, 25% faster than COMA.

Coherence (write) misses are very expensive in COMA 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. As a result, 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; the kernel is the source of the coherence misses.

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

7. The figures show normalized execution time. Speedup is computed based on speed, which is the inverse of execution time[13].

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.

6 Exploration of Parameters

The two results that stand out from the experiments in Section 5 are that the RAC consistently performs well and that SCI and COMA suffer because of handler complexity and the resulting high controller occupancy. 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 SCI and COMA.

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 shared by the OS and the RAC. Therefore, any memory assigned to the RAC cannot be used by the OS to allocate pages. A large statically-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.

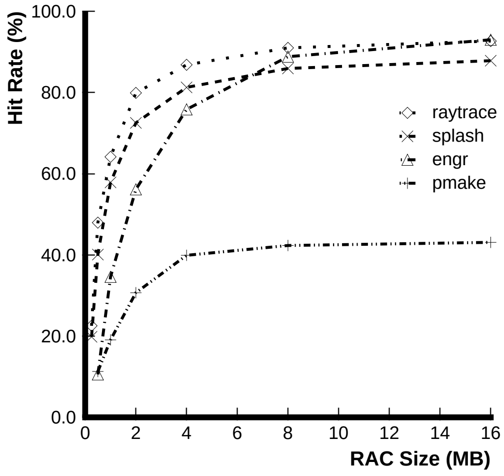

FIGURE 4. Effect of RAC size on RAC hit rate.

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 4 shows the hit rate in the RAC for the different RAC sizes. Figure 5 shows the execution time for each case normalized to DynPtr.

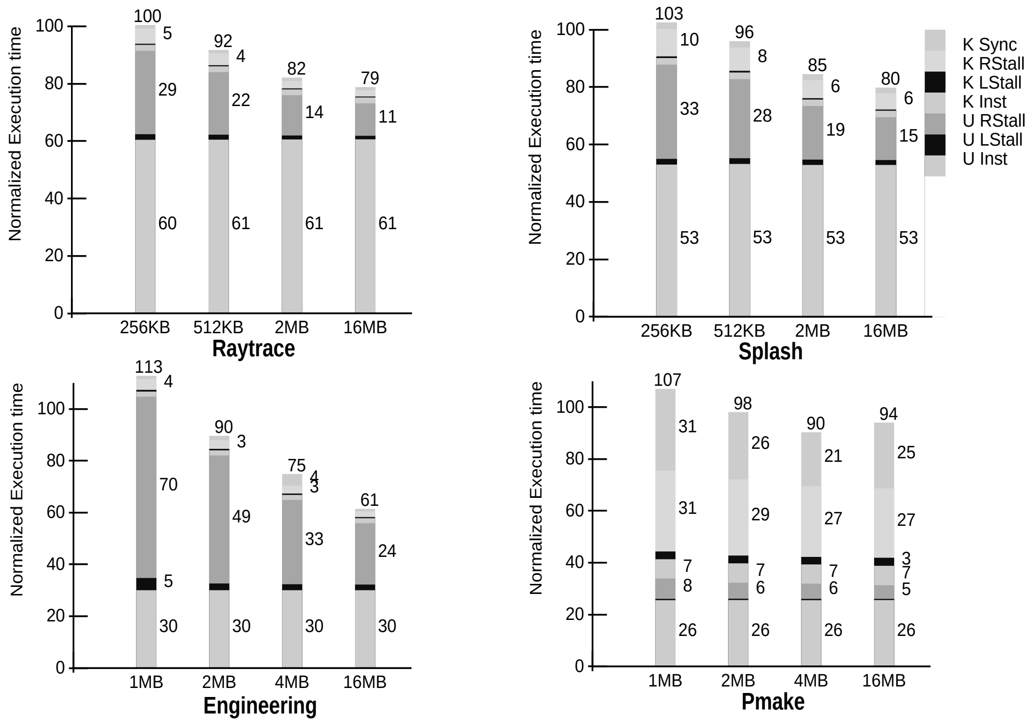

FIGURE 5. Effect of changing RAC size. All times are normalized to the equivalent DynPtr run. The working sets of Raytrace and Splash are smaller than for Pmake and Engineering, so we use smaller RAC sizes**.**

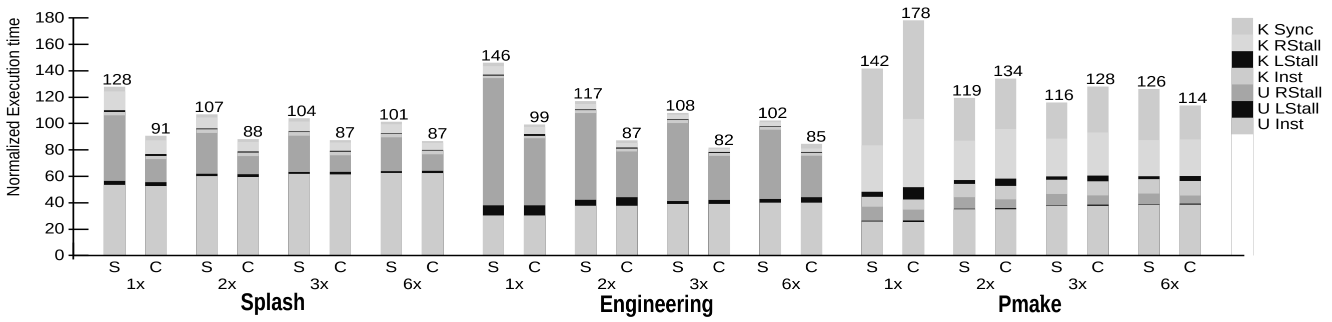

FIGURE 6. Effect of PP speed on SCI (S) and COMA (C) performance. All numbers are normalized to the equivalent DynPtr run with the same PP speed. The 1x PP speed corresponds to the default configuration.

As expected, the performance of the RAC scheme drops as the RAC size is reduced. Figure 4 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). Figure 5 shows 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. 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.

We expect slowdowns of a similar nature for COMA as the reserved memory is reduced.

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 SCI and 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 Pmake. The results of these runs are shown in Figure 6. In each case the SCI and COMA run times are normalized to a DynPtr run with the correspondingly faster PP.

Figure 6 shows that SCI can benefit from a faster node controller, i.e. from lower PP occupancy. SCI's performance relative to DynPtr improves dramatically at the 2x PP speed, but begins to flatten out at 3x and faster speeds. Even with a faster PP, SCI never outperforms DynPtr. SCI still invokes more handlers per miss than DynPtr and has to consult its distributed protocol state at both the requestor and the home. This intrinsic overhead is not affected by increases in controller speed.

We see that the COMA protocol can also 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 other protocols, RAC and MigRep, will see less benefit from a faster PP in most cases because they are not as occupancy-bound as SCI and COMA.

7 MIGRAC: A New Proposal

Our results show that the RAC scheme has many advantages: it captures 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 the following 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.

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-size 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.

We call this new hybrid scheme MIGRAC. We evaluate MIGRAC by comparing its performance to that of the equivalent RAC and MigRep schemes. We size the RAC at a level where it is

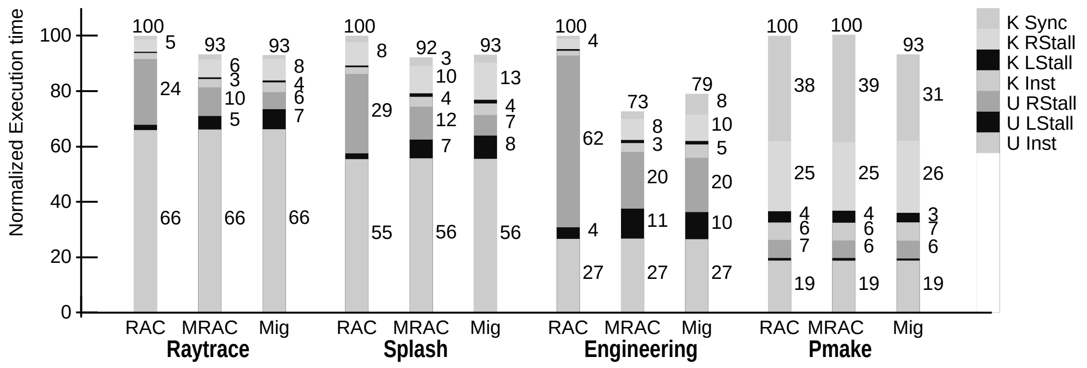

FIGURE 7. Performance of the MIGRAC hybrid scheme. The MIGRAC (MRAC) performance is compared to the equivalent RAC and MigRep (Mig) runs. All times are normalized to the RAC run for that workload. The Pmake runs are normalized to the appropriate MigRep kernel with migration and replication disabled.

showing a drop in performance because of capacity problems (2MB for Pmake, 1MB for Engineering, and 512KB for Splash and Raytrace). A larger RAC size is not interesting because the RAC by itself is able to cache most of the required data. The execution time results are shown in Figure 7 for the three configurations. As in the base workload results, the Pmake time for MigRep and MIGRAC are normalized to runs with the same kernel, but with migration and replication disabled.

The execution time results show that MIGRAC is performing as well as or better than either the equivalent RAC or MigRep runs. The only exception is Pmake, where it performs slightly worse than the equivalent MigRep run. In the other three workloads it performs better than the RAC run, as much as 37% faster for Engineering.

A more subtle advantage of MIGRAC can be seen from the numbers in Table 6. 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. 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.

8 Related Work

Our work is novel in that it compares real implementations of hardware-based and software-based migration/replication schemes on realistic workloads, and includes the effects of OS accesses as well. Most previous studies have focused either exclusively on software-based approaches[3][5][28], or on all-hardware schemes [14][29]. The OS-based page migration and replication work only considered the benefits of better data locality over straight NUMA[5] or CC-NUMA[28] hardware. In [29], Zheng & Torrellas compared RAC versus COMA, and found that RAC outperforms COMA in most cases, unless the sharing pattern is migratory. While our results agree, their methodology and workloads were quite different; their evaluation used scientific workloads and did not use actual implementations of the COMA and RAC schemes. Stenstrom *et al*[14] briefly compare page migration and COMA for scientific workloads and conclude that page migration can perform as well as COMA when sharing patterns are coarse. However, this study used simple estimates for the overhead of migrating or replicating a page. Other studies evaluate RAC implementations in isolation but do not compare them to other mechanisms of migration/replication[18][20]. Hagersten *et al*[10] compare hardware and software-assisted schemes (CC-NUMA, COMA and Simple COMA) 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. This evaluation was done without real protocol implementations, and they also do not consider explicit OS page migration/replication. Falsafi and Wood[7] propose R-NUMA, a hybrid scheme of Simple COMA and CC-NUMA, and compare it to each individually for scientific applications. R-NUMA is found to be more robust than either Simple COMA and CC-NUMA.

Our MIGRAC hybrid is most similar to R-NUMA and Simple COMA. Simple COMA[10] is a hybrid hardware/software 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 a datum is present. Coherence granularity is kept separate from allocation granularity. R-NUMA[7] is a hybrid scheme that dynamically changes the coherence protocol on a per-page basis, switching between Simple COMA and CC-NUMA depending on the sharing pattern. The difference between these two schemes and MIGRAC

TABLE 6. Robustness of MIGRAC.

Workload	RAC hit rate (%)		Pages Replicated	
	RAC	MIGRAC	MigRep	MIGRAC
Raytrace	48.0	55.5	2661	1083
Splash	40.1	50.4	2982	1460
Engr	34.6	46.7	3494	2083
Pmake	25.5	24.8	664	160

is that MIGRAC can perform either replication or migration of a page, while S-COMA and R-NUMA only perform replication: thus, MIGRAC can potentially utilize memory more efficiently by not creating unneeded replicas. Moreover, unlike R-NUMA, MIGRAC uses the same cache coherence protocol for all pages, potentially leading to a simpler implementation.

9 Concluding Remarks

In this paper 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 work is done in the context of the Stanford FLASH multiprocessor, exploiting the fact that the MAGIC memory controller allows one to use the local DRAM in a variety of ways. The schemes studied are (i) base CC-NUMA (DynPtr and SCI), (ii) CC-NUMA+RAC, (iii) COMA, and (iv) CC-NUMA+MigRep. All these schemes are complete and working implementations for the target hardware including operating system modifications, and they work with both user and kernel references from the workloads. As a result, we are able to perform a realistic and fair comparison that has not been available before. Based on the data, we can conclude that:

- SCI's overhead and increased number of handlers invoked per miss are too much to overcome at small machine sizes, and SCI always performs worse than DynPtr. However, as Pmake showed, SCI can improve performance in the face of hot-spotting in the memory system, and may be a better choice in larger-scale machines[12].
- Adding a simple RAC to base CC-NUMA is quite effective in improving performance (e.g., 64% for Engineering workload) by increasing data found in local DRAM. Furthermore, in the case of FLASH, only one additional handler is needed and the changes to other handlers are minor. We find that performance is sensitive to RAC size, but the degradation is typically graceful.
- While COMA can reduce the execution time for some of the applications (e.g., 14% for the Raytrace workload), the additional protocol complexity is substantial (the code size is 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 for the Pmake workload). While protocol processor occupancy (handler complexity) is one of the causes of poor performance, there are also more intrinsic reasons. This is demonstrated by showing that even with a six-times faster protocol processor, Pmake still performs 14% faster with DynPtr than with COMA.
- While MigRep is as effective as RAC for the Engineering workload (56% improvement), in general, it is not as robust in improving performance for some other workloads. MigRep, however, shares the low hardware/firmware implementation complexity with the RAC implementation.
- We show that the MigRep and RAC schemes work in synergistic ways in our hybrid scheme called MIGRAC—the 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.

Finally, looking at our study 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.

Acknowledgments

We would like to thank Shigehiro Asano for his contributions early on in this work. We would also like to thank Edouard Bugnion, Scott Devine and Kinshuk Govil for their help with modifying and debugging the kernel, and Ziyad Hakura and Ulrich Stern for valuable comments on earlier drafts of this paper. We would like to thank Joel Baxter and Robert Bosch for assistance with the SimOS and FlashLite simulators, and Jeffrey Kuskin and David Ofelt for providing FLASH latency values. Finally, we thank Dan Teodosiu for his help with formatting the paper.

This research was supported by DARPA contract DABT63-94-C-0054.

References

[1] Anant Agarwal *et al*. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. *MIT/LCS Memo TM-454*, *Massachusetts Institute of Technology*, 1991.

[2] 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 P*roceedings of the 13th ACM Symposium on Operating System Principles*, pages 95-109, October 1991.

[3] 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.

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

[5] 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 Twelfth ACM Symposium on Operating Systems Principles*, pages 32-43, December 1989.

[6] 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.

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

[8] Steven Frank, Henry Burkhardt III and Dr. James Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. *Compcon '93 Proceedings*.

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

[10] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA Node Implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994.

[11] Mark Heinrich, *et al*. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. October 1994.

[12] M. Heinrich, V. Soundararajan, A. Gupta, and J. Hennessy. A Quantitative Analysis of the Performance and Scalability of Cache Coherence Protocols. Submitted for publication.

[13] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1990.

[14] Truman Joe and John 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

[15] Jeffrey Kuskin, *et al*. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, April 1994.

[16] 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.

[17] D. Lenoski, *et al*. The DASH Prototype: Implementation and Performance. In P*roceedings of the 19th International Symposium on Computer Architecture*, pages 92-103, May 1992.

[18] 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.

[19] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishi. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages I1-I10, August 1995.

[20] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325-336, Chicago, IL, April 1994.

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

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

[23] Richard Simoni. Cache Coherence Directories for Scalable Multiprocessors. Ph.D. Thesis, Technical Report CSL-TR-93-556, Stanford University, November 1992.

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

[25] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 80-91, Gold Coast, Australia, May 1992.

[26] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 12th ACM ACM Symposium on Operating Systems Principles*, pages 159-166, December 1991.

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

[28] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel 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*, October 1996.

[29] Zheng Zhang and Josep 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.