AS-COMA: An Adaptive Hybrid Shared Memory Architecture *

Chen-Chi Kuo, John B. Carter, Ravindra Kuramkote, Mark Swanson

{chenchi, retrac, kuramkot, swanson}@cs.utah.edu
WWW: http://www.cs.utah.edu/projects/avalanche

UUCS-98-010

Department of Computer Science
University of Utah, Salt Lake City, UT 84112

March 23, 1998

Abstract

Scalable shared memory multiprocessors traditionally use either a *cache coherent non-uniform memory access* (CC-NUMA) or *simple cache-only memory architecture* (S-COMA) memory architecture. Recently, hybrid architectures that combine aspects of both CC-NUMA and S-COMA have emerged. In this paper, we present two improvements over other hybrid architectures. The first improvement is a page allocation algorithm that prefers S-COMA pages at low memory pressures. Once the local free page pool is drained, additional pages are mapped in CC-NUMA mode until they suffer sufficient remote misses to warrant upgrading to S-COMA mode. The second improvement is a page replacement algorithm that dynamically backs off the rate of page remappings from CC-NUMA to S-COMA mode at high memory pressure. This design dramatically reduces the amount of kernel overhead and the number of induced cold misses caused by needless thrashing of the page cache. The resulting hybrid architecture is called *adaptive S-COMA* (AS-COMA). AS-COMA exploits the best of S-COMA and CC-NUMA, performing like an S-COMA machine at low memory pressure and like a CC-NUMA machine at high memory pressure. AS-COMA outperforms CC-NUMA under almost all conditions, and outperforms other hybrid architectures by up to 17% at low memory pressure and up to 90% at high memory pressure.

Keywords: Distributed shared memory, multiprocessor computer architecture, memory architecture, CC-NUMA, S-COMA, hybrid.

Technical Areas: Architecture.

*This work was supported by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

1 Introduction

Scalable hardware distributed shared memory (DSM) architectures have become increasingly popular as high-end compute servers. One of the purported advantages of shared memory multiprocessors compared to message passing multiprocessors is that they are easier to program, because programmers are not forced to track the location of every piece of data that might be needed. However, naive exploitation of the shared memory abstraction can cause performance problems, because the performance of DSM multiprocessors is often limited by the amount of time spent waiting for remote memory accesses to be satisfied. When the overhead associated with accessing remote memory impacts performance, programmers are forced to spend significant effort managing data placement, migration, and replication – the very problem that shared memory is designed to eliminate. Thus, the value of DSM architectures is directly related to the extent to which observable remote memory latency can be reduced to an acceptable level.

The two basic approaches for addressing the memory latency problem are building latency-tolerating features into the microprocessor and reducing the average memory latency. Because of the growing gap between microprocessor cycle times and main memory latencies, modern microprocessors incorporate a variety of latency-tolerating features such as fine-grained multithreading, lockup free caches, split transaction memory busses, and out-of-order execution [1, 11, 15]. These features reduce the performance bottleneck of both local and remote memory latencies by allowing the processor to perform useful work while memory is being accessed. However, other than the fine-grained multithreading support of the Tera machine [1], which requires a large amount of parallelism and an expensive and proprietary microprocessor, these techniques can hide only a fraction of the total memory latency. Therefore, it is important to develop memory architectures that reduce the overhead of remote memory access.

Remote memory overhead is governed by three issues: (i) the number of cycles required to satisfy each remote memory request, (ii) the frequency with which remote memory accesses occur, and (iii) the software overhead of managing the memory hierarchy. The designers of high-end commercial DSM systems such as the SUN UE10000 [18] and SGI Origin 2000 [6] have put considerable effort into reducing the remote memory latency by developing specialized high speed interconnects. These efforts can reduce the ratio of remote to local memory latency to as low as 2:1, but they require expensive hardware available only on high-end servers costing hundreds of thousands of dollars. In this paper, we concentrate on the second and third issues, namely reducing the frequency of remote memory accesses while ensuring that the software overhead required to do this remains modest.

Previous studies have tended to ignore the impact of software overhead [5, 12, 16], but our findings indicate that the effect of this factor can be dramatic.

Scalable shared memory multiprocessors traditionally use either a *cache coherent non-uniform memory access* (CC-NUMA) architecture or a *simple cache-only memory architecture* (S-COMA) [16]. Each architecture performs well under different conditions, as follows.

CC-NUMA is the most common DSM memory architecture. It is embodied by such machines as the Stanford DASH [7], SUN UE10000 [18], and SGI Origin 2000 [6]. In a CC-NUMA, shared physical memory is evenly distributed amongst the nodes in the machine, and each page of shared memory has a *home* location. The home node of data can be determined from its global physical address. Processors can access any piece of global data by mapping a virtual address to the appropriate global physical address, but the amount of remote shared data that can be replicated on a node is limited by the size of a node's processor cache(s) and *remote access cache* (RAC) [8]. Thus, CC-NUMA machines generally perform poorly when the rate of conflict or capacity misses is high, such as when a node's caches are too small to hold the entire remote working set or when the data access patterns and cache organization cause cached remote data to be purged frequently.

S-COMA architectures employ any unused DRAM on a node as a cache for remote data [16], which significantly increases the amount of storage available on each node for caching remote data. The performance of pure S-COMA machines is heavily dependent on the *memory pressure* of a particular application. Put simply, memory pressure is a measure of the amount of physical memory in a machine required to hold an application's instructions and data. A 20% memory pressure indicates that 20% of a machine's pages must be used to hold the initial (home) copy of the application's instructions and data. At this low memory pressure, on average 80% of a node's physical memory is available to be used as a page-grained cache of remote data. Although this ability to cache remote data in local memory can dramatically reduce the number of remote memory operations, pure S-COMA has a number of drawbacks. Page management can be expensive. The page-grained allocation of the remote data cache can lead to large amount of internal fragmentation, and the requirement that all shared data accessed by a node *must* be backed by a local DRAM page can lead to thrashing at high memory pressures.

Recently, hybrid architectures that combine aspects of both CC-NUMA and S-COMA have emerged, such as the Wisconsin *reactive CC-NUMA* (R-NUMA) [5] and the USC *victim cache NUMA* (VC-NUMA) [12]. Intuitively, these hybrid systems attempt to map the remote pages for which there are the highest number of conflict misses to local S-COMA pages, thereby eliminating the greatest number of expensive remote operations. All other remote pages are mapped in CC-

NUMA mode. Ideally, such systems would exploit unused available DRAM for caching without penalty but the proposed implementations fail to achieve this goal under certain conditions.

In this paper, we present two improvements over R-NUMA and VC-NUMA. The first improvement is a page allocation algorithm the prefers S-COMA pages at low memory pressures. Once the local free page pool is drained, additional pages are initially mapped in CC-NUMA mode until they suffer sufficient remote misses to warrant upgrading to S-COMA mode. The second improvement is a page replacement algorithm that dynamically backs off the rate of page remappings between CC-NUMA and S-COMA mode at high memory pressure. This design dramatically reduces the amount of kernel overhead and the number of induced cold misses caused by needless thrashing of the page cache. The resulting hybrid architecture is called *adaptive S-COMA* (AS-COMA).

R-NUMA [5] and VC-NUMA [12] initially map all pages in CC-NUMA mode, and then identify remote pages that are suffering inordinate numbers of conflict misses to remote node, so-called *hot* pages. Unfortunately, under heavy memory pressure, there are not enough local pages to accommodate all hot remote pages and thrashing occurs, which severely degrades performance. In addition to the interrupt handling and flushing overheads induced by a remap request, page remapping also increases the cold miss rate, because the contents of both the hot page and any victim page that was downgraded to make room for it must be flushed from the processor cache(s).

AS-COMA initially maps pages in S-COMA mode to exploit S-COMA's superior performance at low memory pressures. Doing so eliminates remote conflict misses and remapping overhead when there is enough free memory to cache all of a node's working set in its local memory. To combat page thrashing under heavy memory pressures, which occurs in S-COMA and to a lesser degree in R-NUMA and VC-NUMA, AS-COMA uses a page replication backoff algorithm to detect thrashing and aggressively reduce its rate of page remapping. Under extreme circumstances, AS-COMA goes so far as to disable CC-NUMA $\leftrightarrow$ S-COMA remappings entirely.

We used detailed execution-driven simulation to evaluate a number of AS-COMA design tradeoffs and then compared the resulting AS-COMA design against CC-NUMA, pure S-COMA, R-NUMA, and VC-NUMA. We found that AS-COMA's hybrid design provides the best behavior of both CC-NUMA and S-COMA. At low memory pressures, AS-COMA acts like S-COMA and outperforms other hybrid architectures by up to 17%. At high memory pressures, AS-COMA avoids the performance dropoff induced by thrashing and aggressively converges to CC-NUMA performance, thereby outperforming the other hybrid architectures by up to 90%. In addition, AS-COMA outperforms CC-NUMA under almost all conditions, and at its worst only underperforms CC-NUMA by 5%.

The remainder of this paper is organized as follows. In Section 2 we describe the basics of all scalable shared memory architectures, followed by an in-depth description of existing DSM models. Section 3 presents the design of our proposed AS-COMA architecture. We describe our simulation environment, test applications, and experiments in Section 4, and present the results of these experiments in Section 5. Finally, we draw conclusions and discuss future work in Section 6.

2 Background

In this section, we discuss organization of the existing DSM architectures: CC-NUMA, S-COMA, R-NUMA, and VC-NUMA.

2.1 Directory-based DSM Architectures

All of the shared memory architectures that we consider share a common basic design, illustrated in Figure 1. Individual nodes are composed of one or more commodity microprocessors with private caches connected to a coherent split-transaction memory bus. Also on the memory bus is a main memory controller with shared main memory and a distributed shared memory controller connected to a node interconnect. The aggregate main memory of the machine is distributed across all nodes. The processor, main memory controller, and DSM controller all snoop the coherent memory bus, looking for memory transactions to which they must respond.

The internals of a typical DSM controller also are illustrated in Figure 1. It consists of a memory bus snooper, a control unit that manages locally cached shared memory (*cache controller*), a control unit that retains state associated with shared memory whose "home" is the local main memory (*directory controller*), a network interface, and some local storage. In all of the design alternatives that we explore, the local storage contains DRAM that is used to store directory state.

When a local processor makes an access to shared data that is not satisfied by its cache, a memory request is put on the coherent memory bus where it is observed by the DSM controller. The bus snooper detects that the request was made to shared memory and forwards the request to the DSM cache controller. The DSM cache controller will then take one of the following two actions. If the data is in main memory, e.g., this node is the memory's "home" or the data is cached in a local S-COMA page, a coherency response is given that allows the main memory controller to satisfy the request. Otherwise the request is forwarded to the appropriate remote node. Once a response has been received, the DSM cache controller supplies the requested data to the processor and potentially also stores it to main memory.

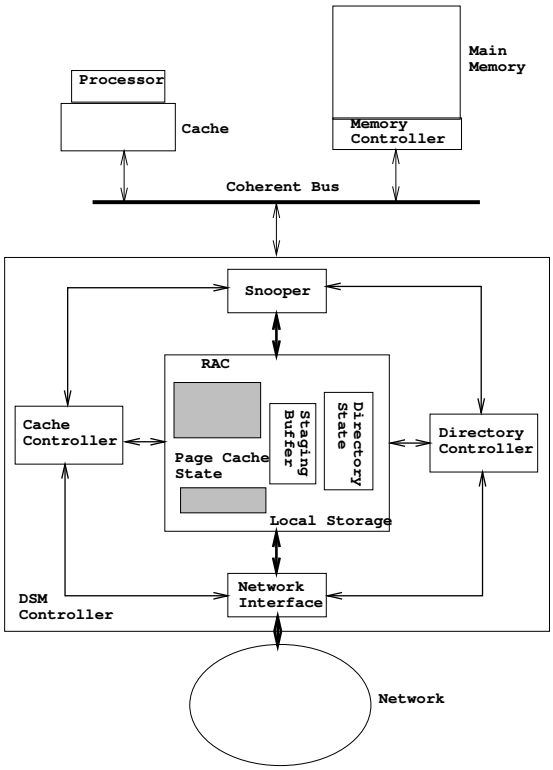

Figure 1 Typical Scalable Shared Memory Architecture

A request for data that is received from a remote node is forwarded to the directory controller, which tracks the status of each line of shared data for which it is the home node. If the remote request can be supplied using the contents of local memory, the directory controller simply responds with the requested data and updates its directory state. If the directory controller is unable to respond directly, e.g., because a remote node has a dirty copy of the requested cache line, it forwards the request to the appropriate node(s) and updates its directory state.

The remote access overhead of these architectures can be represented as:

$(N_{pagecache} * T_{pagecache}) + (N_{remote} * T_{remote}) + (N_{cold} * T_{remote}) + T_{overhead}$.

$N_{pagecache}$ and N_{remote} represent the number of conflict misses that were satisfied by the page cache or remote memory, respectively. N_{cold} represents the number of cold misses induced by flushing and remapping pages, and thus is zero only in CC-NUMA model. $T_{pagecache}$ and T_{remote} represent the latency of fetching the line from the local page cache or remote memory, respectively. $T_{overhead}$ represents the software overheads of the S-COMA and the hybrid models to support page remapping, e.g., flushing.

Table 1 summarizes the remote memory overhead for each architecture and the critical factors determining performance, assuming a fixed amount of memory. Table 2 provides the cost in terms

Model	Remote Overhead	Performance Factors
CC-NUMA	$(N_{remote} * T_{remote})$	Network speed
S-COMA	$(N_{pagecache} * T_{pagecache})+$ $(N_{cold} * T_{remote})+$ $T_{overhead}$	1. Network speed 2. Software overhead
Hybrid Architectures	$(N_{pagecache} * T_{pagecache})+$ $(N_{remote} * T_{remote})+$ $(N_{cold} * T_{remote}) + T_{overhead}$	1. Network speed 2. Software overhead

Table 1 Remote Memory Overhead of Various Models

Model	Storage Cost	Complexity
CC-NUMA	None	None
S-COMA	Page cache state: 1. 2 bits per block 2. 44 bits per page	1.Page cache state lookup 2. local $\leftrightarrow$ remote page map 3. Page-daemon and VM kernel
Hybrid Architectures	Page cache state: 1. 2 bits per block 2. 44 bits per page Refetch Count: 6 bits per page per node	1.Page cache state controller 2. local $\leftrightarrow$ remote page map 3. Page-daemon and VM kernel 4. Refetch counter, comparator and interrupt generator

Table 2 Cost and Complexity of Various Models

of the storage and complexity for each of the models. These issues will be explained in the following sections along with how each model works.

2.2 CC-NUMA

In CC-NUMA, the first page access on each node to a particular page causes a page fault, at which time the local TLB and page table are loaded with a page translation to the appropriate *global* physical page. The home node of each page can be determined from its physical address. When the local processor suffers a cache miss to a line in a remote page, the DSM controller forwards the memory request to the memory's home node, incurring a significant access delay. Remote data can only be cached in the processor cache(s) or an optional remote access cache (RAC) on the DSM controller. Applications that suffer a large number of conflict misses to remote data, e.g., due to the limited amount of caching of remote data, perform poorly on CC-NUMAs [5]. Unfortunately, these applications are fairly common [5, 14, 16]. Careful page allocation [2, 9], migration [21], or replication [21] can alleviate this problem by carefully selecting or modifying the choice of home node for a given page of data, but these techniques have to date only been successful for read-only or non-shared pages.

The conflict miss cost in the CC-NUMA model is represented by $(N_{remote} * T_{remote})$, that is, all misses to shared memory with a remote home must be remote misses. To reduce this overhead, designers of some such systems have adopted high speed interconnect to reduce (T_{remote}) [6, 13, 18].

2.3 S-COMA

In the S-COMA model [16], the DSM controller and operating system cooperate to provide access to remotely homed data. In S-COMA, a mapping from a global virtual address to a *local* physical address is created at the first page fault to that shared memory page. The page fault handler selects an available page from the local DRAM *page cache*. At this time, the cache state information is updated in the local DSM controller to indicate which global page this local page is caching. In addition, the valid bit associated with each cache line in the page is set to invalid to indicate that, while the page mapping is valid, no remote data is actually cached in the local page yet. If there are no free pages in the page cache when a page fault occurs, the page fault handler selects another S-COMA page to replace, flushes this page's cache lines from the local processor cache, and then maps the faulting page.

When a local processor suffers a cache miss to remote data, the DSM cache controller examines the valid bit for the line. If the valid bit is set, the page cache contains valid data for that line, so it can be supplied directly from main memory, thereby avoiding an expensive remote operation. If, however, the requested line is *invalid*, the DSM cache controller must perform a remote request to acquire a copy of the desired data. When the remote node responds with the data, it is written to the page cache, supplied to the processor, and the valid bit is set.

S-COMA's aggressive use of local memory to replicate remote shared data can completely eliminate N_{remote} when the memory pressure on a node is low. However, pure S-COMA's performance degrades rapidly for some applications as memory pressure increases. Because *all* remote data *must* be mapped to a local physical page before it can be accessed, there can be heavy contention if the number of local physical pages available for S-COMA page replication is small. Under these circumstances, thrashing occurs, not unlike thrashing in a conventional VM system. Given the high cost of page replacement, this can lead to dismal performance.

In the S-COMA model, the conflict miss cost is represented by $(N_{pagecache} * T_{pagecache}) + (N_{cold} * T_{remote}) + T_{overhead}$. When memory pressure is low enough that all of the remote data a node needs can be cached locally, page remapping does not occur and both N_{cold} and $T_{overhead}$ are zero. As the memory pressure increases, and thus more remote pages are accessed by a node than can be cached locally, N_{cold} and $T_{overhead}$ increase due to remapping. N_{cold} increases because the contents of any

pages that are replaced from the local page cache must be flushed from the processor cache(s). Subsequent accesses to these pages will suffer cold misses in addition to the cost of remapping. An even worse problem is that as memory pressure approaches 100%, the time spent in the kernel flushing and remapping pages ($T_{overhead}$) skyrockets. Sources of this overhead include the time spent context switching between the user application and the pageout daemon, flushing blocks from the victim page(s), and remapping pages.

2.4 Hybrid DSM Architectures

Two hybrid CC-NUMA/S-COMA architectures have been proposed: R-NUMA [5] and VC-NUMA [12]. We describe these architectures in this section.

The basic architecture of an R-NUMA machine [5] is that of a CC-NUMA machine. However, unlike CC-NUMA, which "wastes" local physical memory not required to hold home pages, R-NUMA uses this otherwise unused storage to cache frequently accessed remote pages, as in S-COMA. This mechanism requires a number of modest modifications to a conventional CC-NUMA's DSM engine and operating system, as described below.

In addition to its normal CC-NUMA operation, the directory controller in an R-NUMA machine maintains an array of counters that tracks for each page the number of times that each processor has refetched a line from that page, as follows. Whenever a directory controller receives a request for a cache line from a node, it checks to see if that node is already in the copyset of nodes for that line. If it is, this request is a *refetch* caused by a conflict miss, and not a coherence or cold miss, and the node's refetch counter for this page is incremented. The per-page/per-node counter is used to determine which CC-NUMA pages are generating frequent remote refetches, and thus are good candidates to be mapped to an S-COMA page on the accessing node. When a refetch counter crosses a configurable threshold (e.g. 64), the directory controller piggybacks an indication of this event with the data response. This causes the DSM engine on the requesting node to interrupt the processor with an indication that a particular page should be remapped to a local S-COMA page.

Pages are remapped from CC-NUMA mode to S-COMA mode using essentially the same mechanism as is used by S-COMA to remap pages. First, all lines of the page being upgraded must be flushed from the local processor cache(s) and RAC. Then, if a free page already exists, the global virtual address is mapped to the selected local physical address, and the DSM engine is informed of the new mapping. If no free page exists, the fault handler first must select a victim page to replace, the victim's data must be flushed from the page cache, and its corresponding global virtual address must be remapped back to its home global physical address.

By supporting both CC-NUMA and S-COMA access modes in the same machine, an R-NUMA machine is able to exploit available local memory as a large page cache for CC-NUMA pages. By tracking refetch counts, it is able to select dynamically which CC-NUMA pages should populate the S-COMA cache based on access behavior. In a recent study [5], R-NUMA's flexibility and intelligent selection of pages to map in S-COMA mode caused it to outperform the best of pure CC-NUMA and pure S-COMA by up to 37% on some applications.

However, although R-NUMA frequently outperforms both CC-NUMA and S-COMA, it was also observed to perform as much as 57% worse on some applications [5]. This poor performance can be attributed to two problems. First, R-NUMA initially maps all pages in CC-NUMA mode, and only upgrades them to S-COMA mode after some number of remote refetches occur, which introduces needless remote refetches when memory pressure is low. Second, R-NUMA always upgrades pages to S-COMA mode when their refetch threshold is exceeded, even if it must evict another hot page to do so. When memory pressure is high, and the number of hot pages exceeds the number of free pages available for caching them, this behavior results in frequent expensive page remappings for little value. This leads to performance worse than CC-NUMA, which never remaps pages.

VC-NUMA [12] treats its RAC as a victim cache for the processor cache(s), i.e., only remote data evicted from the processor cache(s) is placed in its RAC. VC-NUMA reduces memory overhead by using the victim cache tags and page indices to identify the relocation candidates, instead of maintaining multiple refetch counters per page in the directory controller as in R-NUMA. However, this solution requires significant modifications to the processor cache controller and bus protocol, changes that are not feasible in systems built from commodity nodes. The designers of VC-NUMA noticed the tendency of hybrid models to thrash at high memory pressure and suggested a thrashing detection scheme to address the problem. Their scheme requires a local refetch counter per S-COMA page, a programmable *break even* number that depends on the network latency and overhead of relocating pages, and an *evaluation threshold* that depends on the total number of free S-COMA pages in the page cache. Although VC-NUMA frequently outperforms R-NUMA, the study did not isolate the benefit of the thrashing detection scheme from that of the integrated victim cache. Thus, the effectiveness of their thrashing detection scheme under different architecture configurations was not measured and thus the necessity of the extra hardware support was not clearly justified.

In these hybrid models, the conflict miss cost is represented by $(N_{pagecache} * T_{pagecache}) + (N_{remote} * T_{remote}) + (N_{cold} * T_{remote}) + T_{overhead}$. $N_{pagecache}$ and N_{remote} closely depend on the relocation mechanisms. Remappings between CC-NUMA and S-COMA modes account for the increased cold miss rate (N_{cold}), as described earlier. $T_{overhead}$ is the software overhead required for the kernel to handle interrupts, flush pages, and remap pages.

When there are plentiful free local pages, the difference between the hybrid models and S-COMA is that S-COMA does not suffer from as many initial conflict misses, nor does it pay for page remapping. In such a case, the relative costs between the two models can be represented as:

$$N_{remote.hybrid} + N_{cold.hybrid} \gg N_{cold.scoma} \approx 0, \tag{1}$$

$$T_{overhead.hybrid} > T_{overhead.scoma} \approx 0, \tag{2}$$

$$N_{pagecache.scoma} > N_{pagecache.hybrid} \tag{3}$$

As the memory pressure increases, R-NUMA and VC-NUMA suffer from the same problems as pure S-COMA, although to a lesser degree. Even hot pages already in the page cache begin to be remapped. When this occurs, the local page cache becomes less effective at satisfying conflict misses, and $N_{remote.hybrid}+N_{cold.hybrid}$ increases. As before, the extra cold misses are induced by the cache flushes performed during remapping. Also as in S-COMA, as memory pressure approaches 100%, thrashing causes kernel overhead ($T_{overhead.hybrid}$) to become significant. As a result, the performance of the hybrid models drops dramatically under high memory pressure, albeit not as dramatically as pure S-COMA. The primary reason that the hybrids' performance dropoff is less dramatic is that remappings occur only every N (e.g., 64) remote refetches, not on *every* remote access as in S-COMA. In a worst case, the relative cost between the hybrid models and CC-NUMA under high memory pressure can be represented as:

$$N_{remote.hybrid} + N_{cold.hybrid} > N_{remote.ccnuma}, \tag{4}$$

$$T_{overhead.hybrid} \gg T_{overhead.ccnuma} \approx 0. \tag{5}$$

Relations (1), (2) and (3) suggest that one way to improve the hybrid models at low memory pressure is to accelerate their convergence to S-COMA. Likewise, relations (4) and (5) suggest that performance can be improved by throttling CC-NUMA $\leftrightarrow$ S-COMA transitions at high memory pressure. Unlike S-COMA, in which remapping is required for the architecture to operate correctly, the hybrid architectures can choose to stop remapping and leave pages in CC-NUMA mode.

In summary, the performance of hybrid S-COMA/CC-NUMA architectures is significantly influenced by the memory pressure induced by a particular application. Since it is common for users to run the largest applications they can on their hardware, the performance of an architecture at high memory pressures is particularly important. Therefore, it is crucial to conduct performance studies of S-COMA or hybrid architectures across a broad spectrum of memory pressures. An improved hybrid architecture, motivated by the analysis above, that performs well regardless of memory pressure is discussed in the following section.

3 Adaptive S-COMA

At low memory pressure, S-COMA outperforms CC-NUMA, but the converse is true at high memory pressure [16]. Thus, our goal when designing AS-COMA was to develop a memory architecture that performed like pure S-COMA when memory for page caching was plentiful, and like CC-NUMA when it is not.

To exploit S-COMA's superior performance at low memory pressures, AS-COMA initially maps pages in S-COMA mode. Thus, when memory pressure is low, AS-COMA will suffer no remote conflict or capacity misses, nor will it pay the high cost of remapping (i.e., cache flushing, page table remapping, TLB refill, and induced cold misses). Only when the page cache becomes empty does AS-COMA begin remapping.

Like the previous hybrid architectures, AS-COMA reacts to increasing memory pressure by evicting "cold" pages from, and remapping "hot" pages to, the local page cache. However, what sets AS-COMA apart from the other hybrid architectures is its ability to adapt to differing memory pressures to fully utilize the large page cache at low memory pressures and to avoid thrashing at high memory pressures. It does so by dynamically adjusting the *refetch threshold* that triggers remapping, increasing it when it notices that memory pressure is high. If the refetch threshold is too low, remappings will occur too frequently, which leads to thrashing. If it is too high, remappings that could be usefully made will be delayed. By dynamically adjusting the refetch threshold based on both static information (e.g., the cost of relocating a page) and dynamic information (e.g., the rate of page remappings), AS-COMA is able to adapt smoothly to differing memory pressures.

AS-COMA uses the kernel's VM system to detect thrashing, as follows. The kernel maintains a pool of free local pages that it can use to satisfy allocation or relocation requests. The pageout daemon attempts to keep the size of this pool between *free_target* and *free_min* pages. Whenever the size of the free page pool falls below *free_min* pages, the pageout daemon attempts to evict enough "cold" pages to refill the free page pool to *free_target* pages. Only S-COMA pages are considered for replacement. To replace a page, its valid blocks are flushed from the processor cache, and then its corresponding global virtual address is remapped to its home physical address. *Cold* pages are detected using a *second chance* algorithm: the TLB reference bit associated with each S-COMA page is reset each time it is considered for eviction by the pageout daemon. If the reference bit is zero when the pageout daemon next runs, the page is considered cold.

Under low to moderate memory pressure, allocation or relocation requests can be performed immediately because there will be pages in the free page pool. However, at heavy memory pressure, the pageout daemon will be unable to find sufficient cold pages to refill the free page pool. Whenever

the pageout daemon is unable to reclaim at least *free_target* free pages, AS-COMA begins allocating pages in CC-NUMA mode under the assumption that local memory can not accommodate the application's entire working set. In addition, it raises the refetch threshold by a fixed amount to reduce the rate at which "equally-hot" pages in the page cache replace each other. It also increases the time between successive invocations of the pageout daemon. Should the number of hot pages drop, e.g., because of a phase change in the program that causes a number of hot pages to grow cold, the pageout daemon will detect it by detecting an increase in the number of cold pages. At this point, it can reduce the refetch threshold.

Using this backoff scheme, the rate at which destructive flushing and remapping occurs is decreased, as is the number of cold misses induced by remapping. In addition, the frequency at which the pageout daemon is invoked is reduced, which eliminates context switches and pageout daemon execution time. Overall, we found this back pressure on the replacement mechanism to be extremely important. As will be shown in Section 5, it alleviates the performance slowdowns experienced by R-NUMA or VC-NUMA when memory pressure is high.

AS-COMA's conflict miss cost is identical to SCOMA's when there are enough local free pages to accommodate the application's working set. In such cases, the remote refetch cost of AS-COMA will be close to $(N_{pagecache} * T_{pagecache})$. Until memory pressure gets high, N_{rem} will grow slowly. Eventually the page cache will no longer be large enough to hold all hot pages. Ideally AS-COMA's performance would simply degrade smoothly to that of CC-NUMA, $(N_{rem} * T_{rem})$, as memory pressure approaches 100%. Realizable AS-COMA models will fare somewhat worse due to the extra kernel overhead incurred before the system stabilizes. Nevertheless, AS-COMA is able to converge rapidly to either S-COMA or CC-NUMA mode, depending on the memory pressure.

4 Performance Evaluation

4.1 Experimental Setup

All experiments were performed using an execution-driven simulation of the HP PA-RISC architecture called Paint (PA-interpreter)[17, 19]. Paint was derived from the Mint simulator[20]. Our simulation environment includes detailed simulation modules for a first level cache, system bus, memory controller, network interconnect, and DSM engine. It provides a multiprogrammed process model with support for operating system code, so the effects of OS/user code interactions are modeled. The simulation environment includes a kernel based on 4.4BSD that provides scheduling, interrupt handling, memory management, and limited system call capabilities. The modeled physical page size is 4 kilobytes. The VM system was modified to provide the page translation,

allocation, and replacement support needed by the various distributed shared memory models. All three hybrid architectures we study adopt BSD4.4's page allocation mechanism and paging policy [10] with minor modifications. *Free_min* and *free_target* (see Section 3) were set to 5% and 7% of total memory, respectively. We extended the first touch allocation algorithm [9] to distribute home pages equally to nodes by limiting the number of home pages that are allocated at each node to a proportional share of the total number of pages. Once this limit is reached, remaining pages are allocated in a round robin fashion to nodes that have not reached the limit.

The modeled processor and DSM engine are clocked at 120MHz. The system bus modeled is HP's Runway bus, which is also clocked at 120MHz. All cycle counts reported herein are with respect to this clock. The characteristics of the L1 cache, RACs, and network that we modeled are shown in Table 3.

For most of the SPLASH2 applications we studied, the data sets provided have a primary working set that fits in an 8-kbyte cache[22]. We, therefore, model a single 8-kilobyte direct-mapped processor cache to compensate for the small size of the data sets, which is consistent with previous studies of hybrid architectures[5, 12].

We model a 4-bank main memory controller that can supply data from local memory in 58 cycles. The size of main memory and the amount of free memory used for page caching was varied to test the different models under varying memory pressures.

We modeled a sequentially-consistent write-invalidate consistency protocol. DSM data is moved in 128-byte (4-line) chunks to amortize the cost of remote communication and reduce the memory overhead of directory state information. As part of a remote memory access, the DSM engine writes the received data back to the RAC or main memory as appropriate. Our CC-NUMA and hybrid models are not "pure," as we employ a 128-byte RAC containing the last remote data received as part of performing a 4-line fetch. This minor optimization had a larger impact on performance than we had anticipated, as is described in the next section. We do not consider different RAC configurations in the hybrid architectures for this study. An initial relocation threshold of 32,

Component	Characteristics
L1 Cache	Size: 8-kilobytes. 32 byte lines, direct-mapped, virtually indexed, physically tagged, non-blocking, up to one outstanding miss, write back, 1-cycle hit latency
RAC	128 byte lines, direct-mapped, non-inclusive, non-blocking, up to one outstanding miss.
Networks	1 cycle propagation, 2X2 switch topology, port contention (only) modeled Fall through delay: 4 cycles (ratio between remote to local memory access latencies - 3:1)

Table 3 Cache and Network Characteristics

the number of remote refetches required to initiate a page remapping, is used in all three hybrid architectures. The relocation thresholds were incremented by 8 whenever thrashing is detected by AS-COMA's software scheme or by VC-NUMA's hardware scheme; R-NUMA does not employ a backoff scheme. VC-NUMA uses a breakeven number of 16 for its thrashing detection mechanism. We did not simulate VC-NUMA's victim-cache behavior, because we considered the use of non-commodity processors or busses to be beyond the scope of this study. Thus, the results reported for VC-NUMA are only relevant for evaluating its relocation strategy, and not the value of treating the page cache as a victim cache[12].

Finally, Table 4 shows the minimum latency required to satisfy a load or store from various locations in the global memory hierarchy. The average latency in our simulation is considerably higher than this minimum because of contention for various resources (bus, memory banks, networks, etc.), which we accurately model. The remote to local memory access ratio is about 3:1. Note that our network model only accounts for input port contention.

4.2 Benchmark Programs

We used six programs to conduct our study: `barnes`, `fft`, `lu`, `ocean`, and `radix` from the SPLASH-2 benchmark suite [22] and `em3d` from a shared memory implementation of the Split-C benchmark [4, 3]. Table 5 shows the inputs used for each test program. The column labeled *Home pages* indicates the number of shared data pages initially allocated at each node. These numbers indicate that each node manages from 0.5 megabytes (`barnes`) to 2 megabytes (`lu, em3d, and ocean`) of home data.

The *Maximum remote pages* column indicates the maximum number of remote pages that are accessed by a node for each application, which gives an indication of the size of the application's global working set. The *Ideal pressure* column is the memory pressure below which S-COMA and AS-COMA machines act like a "perfect" S-COMA, meaning that every node has enough free memory to cache all remote pages that it will ever access. Below this memory pressure, S-COMA

Data Location	Latency
L1 Cache	1 cycle
Local Memory	58 cycles
RAC	23 cycles
Remote Memory	147 cycles

Table 4 Minimum Access Latency

and AS-COMA never experience a conflict miss to remote data, nor will they suffer any kernel or page daemon overhead to remap pages.

Due to its small default problem size and long execution time, `lu` was run on just 4 nodes - all other applications were run on 8 nodes.

5 Results

Figures 2 and 3 show the performance of CC-NUMA, S-COMA, and three hybrid CC-NUMA/S-COMA architectures (AS-COMA, VC-NUMA, R-NUMA) on the six applications. The left column in each figure displays the execution time of the various architectures relative to CC-NUMA, and indicates where this time was spent by each program[1]. The right column in each figure displays where cache misses to shared data were satisfied[2]. Note that for readability, these graphs are adjusted to focus on the remote data accesses, and thus the origin of the Y-axis is non-zero. We simulated the applications across a range of memory pressures between 10% and 90%. Only one result is shown for CC-NUMA, since it is not affected by memory pressure. As can be seen in the graphs, the relative performance of the different architectures can vary dramatically as memory pressures change. All results include only the parallel phase of the various programs.

Program	Input parameters	Home Pages (per node)	Maximum Remote Pages	Ideal Pressure
`barnes`	16K particles	102	552	16
`em3d`	40K nodes,15%remote, 20 iters	491	778	39
`FFT`	256K Points, tuned for cache sizes	390	1254	24
`LU`	1024x1024 matrix, 16x16 blocks, contiguous	514	405	56
`ocean`	258x258 ocean	473	356	57
`radix`	1M Keys, Radix = 1024	259	1306	17

Table 5 Programs and Problem Sizes Used in Experiments

[1] *U-SH-MEM*: stalled on shared memory. *K-BASE*: performing essential kernel operations (i.e., those required by all architectures). *K-OVERHD*: performing architecture-specific kernel operations, such as remapping pages and handling relocation interrupts. *U-INSTR* and *U-LC-MEM*: performing user-level instructions or non-shared memory operations. *SYNC*: performing synchronization operations.

[2] *HOME*: the local node is the data's home, so it is supplied from local DRAM. *S-COMA*: misses satisfied from the *local* page cache. *RAC*: misses satisfied from the *local* RAC. *COLD*: cold misses satisfied on a *remote* node, including both essential cold misses and cold misses induced by remapping. *CONF/CAPC*: conflict/capacity misses not satisfied locally but that instead result in remote accesses.

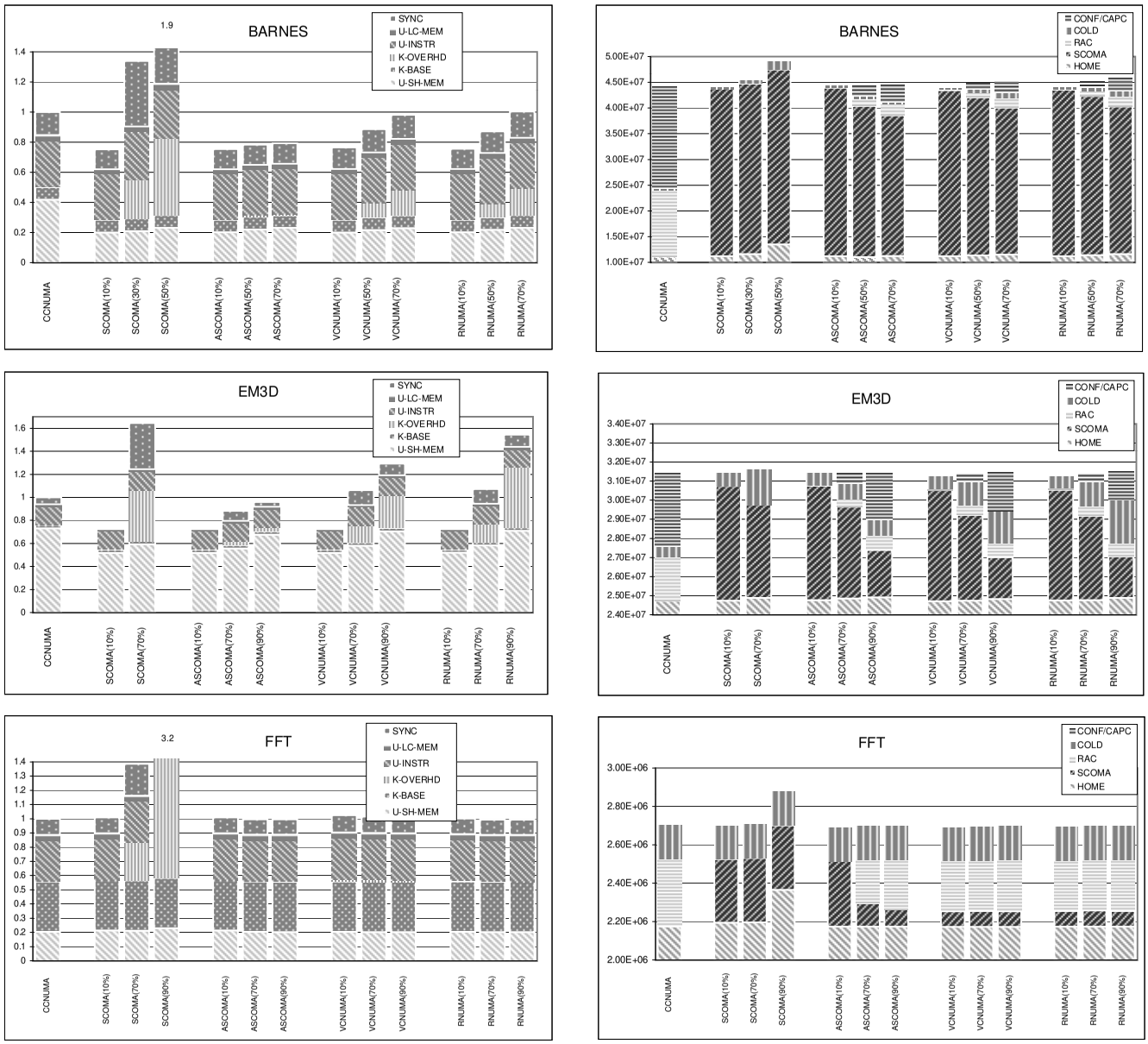

Figure 2 Performance Charts for `barnes`, `em3d` and `fft`. (Left: Relative Execution Time. Right: Where Misses Were Satisfied)

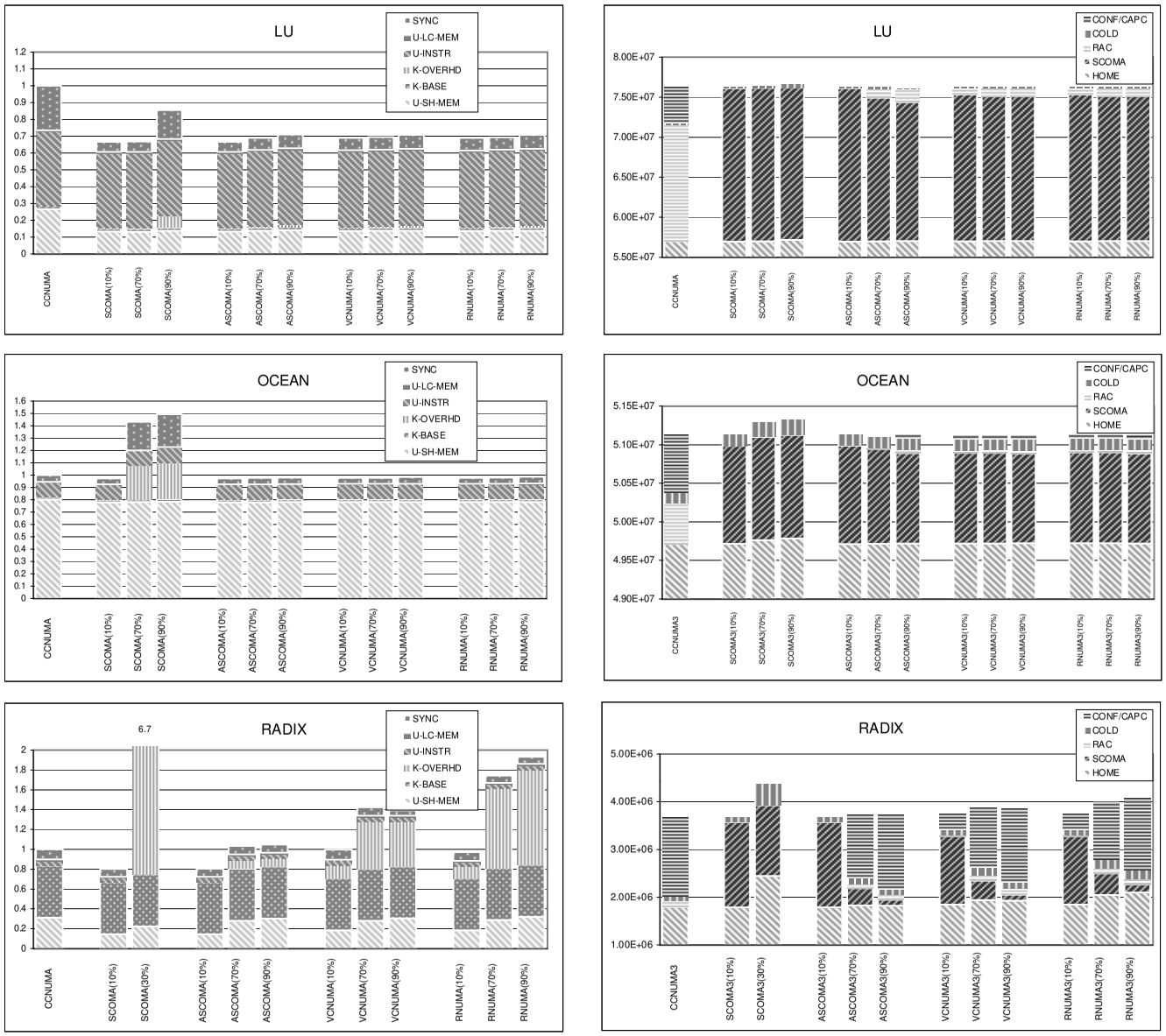

Figure 3 Performance Charts for `lu`, `ocean`, and `radix`. (Left: Relative Execution Time. Right: Where Misses Were Satisfied)

5.1 Initial Allocation Schemes

We will first focus on the effect of the initial allocation policies. Recall from Table 5 that the "ideal" memory pressure for the six applications ranged from 16% to 57%. Below this memory pressure, the local page cache is large enough to store the entire working set of a node. To isolate the impact of initially allocating pages in S-COMA, we simulated S-COMA and the hybrid architectures at a memory pressure of 10%, when no page remappings beyond any initial ones will occur. Table 6 shows the percentage of remote pages that are refetched at least 32 times, and thus will be remapped from CC-NUMA to S-COMA mode in R-NUMA or VC-NUMA, versus of the total number of remote pages accessed. This percentage exhibits a broad range from under 1% in `fft` to over 95% in `lu` and `radix`.

First, to illustrate the importance of employing a hybrid memory architecture over a vanilla CC-NUMA architecture, examine their relative results at 10% memory pressures, in Figures 2 and 3. Under these circumstances, AS-COMA, like S-COMA, outperforms CC-NUMA by 20-35% for four of the applications (`lu`, `radix`, `barnes`, and `em3d`). Looking at the hybrid architectures in isolation, we can see that for `radix`, AS-COMA outperforms R-NUMA and VC-NUMA by 17%. In `radix`, the percentage and total number of remote pages that need to be remapped are both quite high, 98% and 10236 respectively. In the other applications, the initial page allocation policy had little impact on performance. There is no strong correlation between the number of pages that need to be remapped and performance. We can observe a 5% performance benefit in `lu`, where the percentage of relocated remote pages is very high (99%), but the total number is fairly small (1606).

There are two primary reasons why the initial allocation policy did not have a stronger impact on performance. First, our interrupt and relocation operations are highly optimized, requiring only 2000 and 6000 cycles, respectively, to perform. Thus, the impact of the unnecessary remappings and flushes is overwhelmed by other factors. Second, as an artifact of our experimental setup,

Program	Total Remote Pages	Relocated Pages	% of Relocated Pages
`barnes`	4416	3498	80%
`em3d`	6224	1868	29%
`FFT`	10032	5	0.05%
`LU`	1620	1606	99%
`ocean`	2848	569	20%
`radix`	10448	10236	98%

Table 6 Number of Remote Pages Ever Accessed versus Conflicted Frequently

the initial remappings for several applications were not included in the performance results, as they took place before the parallel phase when our measurements are taken. This was the case for `barnes` and `em3d`. The final two applications, `fft` and `ocean`, only access a small number of remote pages enough times to warrant remapping, and thus the impact of initially mapping pages in S-COMA mode is negligible.

In summary, if memory pressure is low and local pages for replication are abundant, an S-COMA-preferred initial allocation policy can improve the performance hybrid architectures moderately by accelerating their convergence to pure S-COMA behavior. However, the performance boost is modest.

5.2 Thrashing Detection and Backoff Schemes

The performance of hybrid DSM architectures depends heavily on the memory pressure. Performance seriously degrades when the page cache cannot hold all "hot" pages and those pages start to evict one another. Intuitively, when this begins to occur, the memory system should simply treat the page cache as a place to store a *reasonable* set of hot pages, and stop trying to fine tune its contents since this tuning adds significant overhead. Previous studies have not considered the kernel overhead ($T_{overhead}$), but we found it to be very significant at high memory pressures. Once the page cache holds only hot pages, further attempts to refine its contents lead to thrashing, which involves unnecessary flushing of hot data, cache flushes, and induced cold misses. Since one hot page is replacing another, the benefit of this remapping is likely to be minimal compared to the cost of the remapping itself. As a result, the performance of a hybrid architecture will quickly drop below that of CC-NUMA if a mechanism is not put in place to avoid thrashing. As described in Section 3, the pageout daemon in AS-COMA detects thrashing when it cannot find cold pages to replace, at which point it reduces the rate of page remappings, going so far as to stop it completely if necessary. As can be seen in Figures 2 and 3, this can lead to significant performance improvements compared to R-NUMA and VC-NUMA under heavy memory pressure.

We can divide the six applications into two groups: (i) applications where there are sufficient remote conflict misses that handling thrashing effectively can lead to large performance gains (`barnes`, `em3d`, and `radix`), and (ii) applications in which minimal efforts to avoid thrashing are sufficient for handling high memory pressure (`fft`, `ocean`, and `lu`).

The behavior of `em3d` shows the danger of focusing solely on reducing remote conflict misses when designing a memory architecture. As shown in Figure 2, the performance of `em3d` on the hybrid architectures is quite sensitive to memory pressure. R-NUMA outperforms CC-NUMA until memory pressure approaches 70%, after which time its performance drops quickly. CC-NUMA

outperforms R-NUMA by 5% at 70% memory pressure and by 50% at 90%. Looking at the detailed breakdown of where time is spent, we can see that increasing kernel overhead is the culprit. In `em3d`, approximately 29% of remote pages, i.e., 230 pages, are eligible for relocation (see Table 6), but at 70% memory pressure there are only 210 free local pages. It turns out that for `em3d`, most of the remote pages ever accessed are in the node's working set, i.e., they are "hot" pages. Thus, above 70% memory pressure, R-NUMA begins to thrash and its performance degrades badly. Looking at the right column of Figure 2, we can see that this performance dropoff occurs even though there are significantly fewer remote conflict misses (CONF/CAPC) in R-NUMA than in CC-NUMA or AS-COMA. The cost of constantly remapping pages between CC-NUMA and S-COMA mode and the increase in remote cold misses overwhelms the benefit of the reduced number of remote conflict misses. This behavior emphasizes the importance of detecting thrashing and reducing the rate of remappings when it occurs.

Recognizing this problem, VC-NUMA uses extra hardware to detect thrashing. However, its mechanism is not as effective as AS-COMA's. VC-NUMA starts to underperform CC-NUMA at the same memory pressure that R-NUMA does, 70%. While VC-NUMA outperforms R-NUMA by 22% at 90% memory pressure, it underperforms CC-NUMA by 27% and AS-COMA by 31%. In contrast, AS-COMA outperforms CC-NUMA even at 90% memory pressure, when the other hybrid architectures are thrashing. It does so by dynamically turning off relocation as it determines that this relocation has no benefits because it is simply replacing hot pages with other hot pages. This results in more remote conflict/capacity misses than the other hybrid architectures, but it reduces the number of cold misses caused by flushing pages during remapping and the kernel overhead associated with handling interrupts and remapping. As a result, AS-COMA outperforms VC-NUMA by 31% and R-NUMA by 53% at 90% memory pressure. Moreover, despite having only a small page cache available to it and a remote working set larger than this cache, AS-COMA outperforms CC-NUMA.

`Barnes` exhibits very high spatial locality. It accesses large dense regions of remote memory, and thus can make good use of a local S-COMA page cache[3]. As shown in in Table 5, `barnes`'s ideal memory pressure is 16%. Like `em3d`, most of the remote pages that are accessed are part of the working set and "hot" for long periods of execution. We observed that thrashing begins to occur at 50% memory pressure. As in `em3d`, R-NUMA reduces the number of remote conflict/capacity misses at high memory pressures, at the cost of increasing kernel overhead and remote cold misses.

[3]Note that `barnes` is very compute-intensive, and a problem size that can be simulated in a reasonable amount of time requires only approximately 100 home pages per node of data. Since there are only about 50 free pages per node available for page replication at 70% memory pressure, we did not simulate `barnes` at higher memory pressures since the results would be heavily skewed by small sample size effects.

As a result, it is able to outperform CC-NUMA at low memory pressure, but is only able to break even by the time memory pressure reaches 70%. Similarly, VC-NUMA's backoff mechanism is not sufficiently aggressive at moderate memory pressures to stop the increase in kernel overhead or cold misses. In particular, VC-NUMA only checks its backoff indicator when an average of two replacements per cached page have occurred, which is not sufficiently often to avoid thrashing. As shown in the previous study [12], VC-NUMA does not significantly outperform R-NUMA until memory pressure exceeds 87.5%. Once again, AS-COMA's adaptive replacement algorithm detects thrashing as soon as it starts to occur, and the resulting backoff mechanism causes performance to degrade only slightly as memory pressure increases. As a result, it consistently outperforms CC-NUMA by 20% across all ranges of memory pressures, and outperforms the other hybrid architectures by a similar margin at high memory pressures.

Unlike `barnes`, `radix` exhibits almost no spatial locality. Every node accesses every page of shared data at some time during execution. As such, it is an extreme example of an application where fine tuning of the S-COMA page cache will backfire - each page is roughly as "hot" as any other, so the page cache should simply be loaded with some reasonable set of "hot" pages and left alone. With an ideal memory pressure of 17% and low spatial locality, the performance of pure S-COMA is 6.7 times worse than CC-NUMA's at memory pressures as low as 30%. Although the performance of both R-NUMA and VC-NUMA are significantly more stable than that of S-COMA, they too suffer from thrashing by the time memory pressure reaches 70%. The source of this performance degradation is the same as in `em3d` and `barnes` - increasing kernel overhead and (to a lesser degree) induced cold misses. Once again, R-NUMA induces fewer remote accesses than CC-NUMA, but the kernel overhead required to support page relocation is such that R-NUMA underperforms CC-NUMA by 75% at 70% memory pressure and by almost a factor of two at 90% memory pressure. Once again, VC-NUMA's backoff algorithm proves to be more effective than R-NUMA's, but it still underperforms CC-NUMA by roughly 40% at high memory pressures. AS-COMA, on the other hand, deposits a reasonable subset of "hot" pages into the page cache and then backs off from replacing further pages once it detects thrashing. As a result, even for a program with almost no spatial locality, AS-COMA is able to converge to CC-NUMA-like performance (or better) across all memory pressures. At 90% memory pressure, AS-COMA outperforms VC-NUMA by 35% and R-NUMA by 90% at high memory pressures, and it remains within 5% of CC-NUMA's performance. The slight degradation compared to CC-NUMA is due to the short period of thrashing that occurs before AS-COMA can detect it and completely stop relocations.

Applications in the second category (`fft`, `ocean`, and `lu`) exhibit good page-grained locality. All three applications only have a small set of "hot" pages, which can be easily replicated using

a small page cache, or references to remote pages are so localized that the small (128-byte) RAC in our simulation was able to satisfy a high percentage of remote accesses. As a result, thrashing never occurs and the various backoff schemes are not invoked. Thus, the performance of the three hybrid algorithms is almost identical.

The performance results for `fft` and `ocean` are almost identical, albeit for different reasons. For these applications, all of the architectures performed equally well, except for pure S-COMA, which performs poorly at high memory pressures. As can be seen in Table 6, only a tiny fraction of pages in `fft` are accessed enough to be eligible for relocation, so all of the hybrid architectures effectively become CC-NUMAs. S-COMA must maintain inclusion between the processor cache and the page cache, so kernel overhead due to thrashing occurs at 90% memory pressure, which causes performance to drop significantly. Somewhat surprisingly, `fft` has such high spatial locality in its references to remote memory that the 128-byte RAC plays a major role in satisfying remote accesses locally. The reason that performance is stable across all memory pressures in `ocean` can be seen in the right hand graph of Figure 3. Even at 90% memory pressure, only 3% of cache misses are to remote data, and most such accesses can be supplied from a local S-COMA page or the RAC. As a result, all of the architectures other than pure S-COMA, which suffers the same problem as in `fft`, perform within 3% of one another.

Finally, in `lu`, each process accesses every remote page enough times to warrant remapping (see Table 6), similar to `radix`. However, every process uses each set of shared pages in the problem set for only a short time before moving to another set of pages. Thus, unlike `radix`, only a small set of remote pages are active at any time, and a small page cache can hold each process's active working set completely. So, while 7% of CC-NUMA's cache misses must be satisfied by remote nodes, practically all cache misses are satisfied locally in the other architectures. As a result, all of the hybrid architectures outperform CC-NUMA by approximately 33% across all memory pressures. Even pure S-COMA outperforms CC-NUMA at a 90% memory pressure, although its overall performance is 15% worse than the hybrid architectures because of load imbalance.

In summary, for applications that do not suffer frequent remote cache misses or for which the active working set of remote pages is small at any given time, all of the hybrid architectures perform quite well, often outperforming CC-NUMA. However, for applications with less spatial locality or larger working sets, the more aggressive remapping backoff mechanism used by AS-COMA is crucial to achieving good performance. In such applications, AS-COMA outperformed the other hybrid architectures by 20% to 90%, and either outperformed or broke even with CC-NUMA even at extreme memory pressures. Given programmers' desire to run the largest problem size that they

can on their machines, this stability of AS-COMA at high memory pressures could prove to be an important factor in getting hybrid architectures adopted.

6 Conclusions

The performance of hardware distributed shared memory is governed by three factors: (i) remote memory latency, (ii) the number of remote misses, and (iii) the software overhead of managing the memory hierarchy. In this paper, we evaluated the performance of five DSM architectures (CC-NUMA, S-COMA, R-NUMA, VC-NUMA, and AS-COMA) with special attention to the third factor, system software overhead. Furthermore, since users of SMPs tend to run the largest applications possible on their hardware, we paid special attention to how well each architecture performed under high *memory pressure*.

We found that at low memory pressure, architectures that were most aggressive about mapping remote pages into the local page cache (S-COMA and AS-COMA) performed best. In our study, S-COMA and AS-COMA outperformed the other architectures by up to 17% at low memory pressures. As memory pressure increased, however, it became increasingly important to reduce the rate at which remote pages were remapped into the local page cache. S-COMA's performance usually dropped dramatically at high memory pressures. The performance of VC-NUMA and R-NUMA also dropped at high memory pressures, albeit not as severely as S-COMA, due to thrashing. This thrashing phenomenom has been largely ignored in previous studies, but we found that it had a significant impact on performance, especially at the high memory pressures likely to be preferred by power users.

In contrast, AS-COMA's software-based scheme to detect thrashing and reduce the rate of page remappings caused it to outperform VC-NUMA and R-NUMA by up to 90% at high memory pressures. AS-COMA is able to fully utilize even a small page cache by mapping a subset of "hot" pages locally, and then backing off further remapping. This mechanism caused AS-COMA to outperform even CC-NUMA in five out of the six applications we studied, and only underperform CC-NUMA by 5% in the sixth.

Consequently, we believe that hybrid CC-NUMA/S-COMA architectures can be made to perform effectively at all ranges of memory pressures. At low memory pressures, aggressive use of available DRAM can eliminate most remote conflict misses. At high memory pressures, reducing the rate of page remappings and keeping only a subset of "hot" pages in the small local page cache can lead to performance close to or better than CC-NUMA. To achieve this level of performance,

the overhead of system software must be carefully considered, and careful attention must given to avoiding needless system overhead. AS-COMA achieves these goals.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, September 1990.

[2] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[3] S. Chandra, J.R. Larus, and A. Rogers. Where is time spent in message-passing and shared-memory programs? In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, October 1994.

[4] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in spit-c. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[5] B. Falsafi and D.A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[6] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *SIGARCH97*, pages 241–251, June 1997.

[7] 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.

[8] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[9] M. Marchetti, L. Kontothonassis, R. Bianchini, and M.L. Scott. Using simple page placement policies to reduce the code of cache fills in coherent shared-memory systems. In *Proceedings of the Ninth ACM/IEEE International Parallel Processing Symposium (IPPS)*, April 1995.

[10] M.K. Mckusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4BSD operating system*, chapter 5 Memory Management, pages 117–190. Addison-Wesley Publishing Company Inc, 1996.

[11] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, December 1996.

[12] 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.

[13] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp scalable shared memory multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, 1995.

[14] S. E. Perl and R.L. Sites. Studies of Windows NT performance using dynamic execution traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 169–184, October 1996.

[15] V. Santhanam, E.H. Fornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.

[16] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for Simple COMA. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 276–285, January 1995.

[17] L.B. Stoller, R. Kuramkote, and M.R. Swanson. PAINT- PA instruction set interpreter. Technical Report UUCS-96-009, University of Utah - Computer Science Department, September 1996.

[18] Sun Microsystems. Ultra Enterprise 10000 System Overview. http://www.sun.com/servers/datacenter/products/starfire.

[19] M. Swanson and L. Stoller. Shared memory as a basis for conservative distributed architectural simulation. In *Parallel and Distributed Simulation (PADS '97)*, 1997. Submitted for publication.

[20] J.E. Veenstra and R.J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *MASCOTS 1994*, January 1994.

[21] 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 Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[22] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.