Multiprocessor Cache Coherence Based on Virtual Memory Support

Karin Petersen
Computer Science Laboratory
Xerox PARC

Kai Li
Department of Computer Science
Princeton University

- **Proposed running head:**
  Virtual Memory based Cache Coherence

- **Contact Address:**
  Karin Petersen
  Xerox PARC
  3333 Coyote Hill Road
  Palo Alto, CA 94304

- **Abstract:**

  Virtual memory based cache coherence is a mechanism that relies only on hardware that already exists on the microprocessors of a shared memory multiprocessor system, yet dynamically detects and resolves potential cache inconsistencies using virtual-memory techniques. The key feature of the approach is that the virtual memory translation hardware on each processor is used to detect shared accesses that could lead to memory incoherencies, and VM page fault handlers execute the appropriate actions to maintain cache coherence.

  VM-based cache coherence basically trades off design simplicity against increased software overheads. The work presented in this paper evaluates this tradeoff. We show that VM-based cache coherence performs well for scientific applications that require significant aggregate memory bandwidth.

- **Keywords:** shared memory, multiprocessors, cache coherence, virtual memory, performance evaluation.

- **Biographies:**

  Karin Petersen is a Member of the Research Staff at Xerox PARC. She got her Ph.D. from Princeton University in 1993 and joined the Computer Science Laboratory at PARC that same year. Dr. Petersen's research interests include mobile, distributed and parallel systems, parallel architectures, performance evaluation, and user interfaces.

  Kai Li is an Associate Professor of Computer Science at Princeton University. He got his Ph.D. from Yale University in 1986 and joined Princeton immediately thereafter. Prof. Li's research interests are operating systems, computer architecture and parallel computing. Currently he is leading the SHRIMP project, which focuses on building high-performance multicomputers from commodity desktop systems such as PCs, off-the-shelf networks, and uniprocessor operating systems.

List of symbols:

- P_x : Capitalized letter P with x as the subscript.
- $PTE_{x,i}$: The three capitalized letters PTE with x comma i as a subscript.
- $PTE_{self,i}$: The three capitalized letters PTE with the word 'self' followed by a comma and letter i as a subscript.
- $TLB_{x,i}$: The three capitalized letters TLB with x comma i as a subscript.
- $TLB_{self,i}$: The three capitalized letters TLB with the word 'self' followed by a comma and the letter i as a subscript.
- $weak_i$: The four letters 'weak' with i as the subscript.

List of Figures

1 VM-based algorithm for SC 11
2 Transition diagram for sequential consistency. 12
3 VM-based algorithm for RC 13
4 Legend used in graphs. 15
5 Performance of the VM-based schemes. 16
6 Performance with crossbar interconnections. 19
7 Effect of page size on VM-RC performance. 21
8 VM-RC using write-back for shared data 22
9 VM-RC performance with write-back caches and UpdateLists. 23
10 Relative performance with varying memory latencies. 24
11 Relative performance with varying data cache sizes. 25
12 Effect of micro-kernel overheads. 27
13 Effect of slow page based cache line invalidations. 28
14 Performance with larger number of processors. 30

List of Tables

I Simulation Parameters 14
II VM-RC Overhead. 18

Multiprocessor Cache Coherence Based on Virtual Memory Support

Karin Petersen
Computer Science Laboratory
Xerox PARC

Kai Li
Computer Science Department
Princeton University

Abstract

Virtual memory based cache coherence is a mechanism that relies only on hardware that already exists on the microprocessors of a shared memory multiprocessor system, yet dynamically detects and resolves potential cache inconsistencies using virtual-memory techniques. The key feature of the approach is that the virtual memory translation hardware on each processor is used to detect shared accesses that could lead to memory incoherencies, and VM page fault handlers execute the appropriate actions to maintain cache coherence.

VM-based cache coherence basically trades off design simplicity against increased software overheads. The work presented in this paper evaluates this tradeoff. We show that VM-based cache coherence performs well for scientific applications that require significant aggregate memory bandwidth.

1 Introduction

Globally shared memory is an effective paradigm for programming multi-processor machines. However, the level at which the paradigm gets implemented depends on the architecture of the multiprocessor. Architectures that lack support for direct, fine grain access to remote memory modules, such as the Intel Paragon and networks of workstations, require that the shared memory model be fully implemented in software [32, 12]. On the other hand, architectures with memory-mapped network interfaces [42, 28] provide a basis for several different approaches to support the efficient use of the globally shared physical memory. One important feature an efficient hardware-based implementation needs to include is support for per-processor caching of the whole address space, thus requiring mechanisms to ensure consistency between cached copies of shared data.

The most common architecture of shared-memory multiprocessors connects processors, their caches, and the memory modules with a bus. The per-processor caches are kept coherent using snooping protocols implemented in the cache controllers [4]. This approach has been very effective for building small scale multiprocessors [19, 42].

The processor caches in a bus-based architecture can reduce contention for the bus and aide in hiding the processors' memory access latencies. The main drawback of the bus-based multiprocessor architecture is that the bus interconnect can become a bottleneck as the number of processors increases. As the performance gap between microprocessors and DRAMs is widening, the bus-based architectures become even more sensitive to available memory bus bandwidth. This is particularly true for programs that do not exhibit high degrees of data reference locality.

Alternative approaches to building shared-memory architectures use crossbars or routing networks to provide high connectivity and aggregate bandwidth. For these architectures snooping techniques cannot be used to enforce cache consistency since they require atomic

broadcast capabilities to invalidate or update cache blocks. Instead, when provided, cache-coherence is maintained through directory-based protocols implemented in hardware or by relying on compiler support to insert cache flush and invalidation instructions into application code. The hardware-implemented directory protocols require expensive and complicated hardware support, while the compiler-based approaches can only make pessimistic assumptions about cache block usage. We have studied an intermediate approach: a cache consistency protocol that relies only on the memory mapped network interfaces and hardware that already exists on the processors, yet dynamically detects and resolves potential cache inconsistencies using virtual-memory techniques.

The virtual-memory based approach to cache coherence requires that the virtual memory translation hardware of each processor in the system trap memory access attempts that might leave caches incoherent and then uses the virtual memory management software to enforce coherence. This paper describes these mechanisms and presents the results of a study intended to evaluate the tradeoff between the cost of the virtual-memory based mechanisms, which include uncached remote memory accesses to protocol-specific data structures, and the benefit obtained from caching the application's data structures without the need for complicated specialized hardware.

The evaluation of the virtual-memory based protocols is based on trace driven simulations of several scientific applications. We compare the VM-based approach with a popular hardware-based snooping cache approach. The simulation results show that the VM-based mechanisms implemented on a medium-scale crossbar-based multiprocessor can outperform the hardware approach of a bus-based multiprocessor, because of the increased aggregate memory bandwidth. We also show how the performance of the VM-based approach is affected by several parameters such as, cache size, page size, write policies, memory latencies, interrupt latencies, and scale. Although the algorithms presented in this paper are limited to medium scale multiprocessors, Kontothanassis and Scott [27] have recently extended our work to support larger scale machines.

2 Traditional Solutions to the Cache Coherence problem

As mentioned above, most previously proposed and implemented methods for maintaining cache coherence on a shared-memory multiprocessor fall into two categories: hardware-based approaches, that require complete architectural support to implement the protocols, and application-level software methods based entirely on compiler technology. More recently, these mechanisms have been extended to include notions of different degrees of memory consistency, thus allowing increased flexibility in the ordering of the operations required to maintain coherence.

2.1 Hardware Solutions

The most common hardware-based solutions use either snoopy caches [21, 35, 38, 41] or directories [45, 5]. Snoopy-cache protocols rely on a bus-based interconnection of the nodes in the system. Examples of bus based shared memory multiprocessors are the DEC Firefly [46], Sequent Symmetry [33], SGI Power Series [42], and Sun SPARCcenter 2000 [19]. Cache controllers listen to all transactions on the bus and take appropriate actions to keep caches coherent. Snoopy cache architectures do not scale well because the bus traffic increases with the number of processors. More recently snoopying protocols have been implemented on multiprocessors interconnected with slotted ring networks, such as the KSR1 [36]. The

slotted ring topology has much higher bandwidth than available on bus-based interconnection. However, the latency of shared memory accesses using a snoopying protocol on a ring-based interconnection is very high, since the access has to make it around the whole ring, or hierarchy of rings, before it completes.

Directory based cache coherence protocols do not depend on a specific interconnection topology. A directory-based protocol requires that a directory be maintained at each memory module. The directory keeps track of the cached copies of each block from the memory module. When the directory receives an update request for a memory block, it sends an invalidation or update request to all caches with a copy of the block. Examples of recent directory-based shared memory multiprocessors are the Stanford DASH [31] and FLASH [28], the MIT Alewife [11], and the Wisconsin Typhoon [40]. Since directory-based schemes allow arbitrary interconnection networks, they scale better than the bus-based snoopy cache schemes. However, all hardware-based methods suffer from the same problem: they require special purpose hardware that is expensive both in design time and chip real estate. The hardware overhead of the DASH machine was about 30% [31]; the FLASH architecture, as well as other recent multiprocessor architectures, include a second processor to handle all interprocessor communication.

Several research groups have studied alternatives to simplify the hardware implementations of directory protocols. Chaiken et al present a hardware directory protocol that uses software techniques to implement a fully mapped directory, requiring only a constant amount of directory memory [11, 10]. LimitLESS directories implement a small number of copyset pointers for each memory block and trap into software to manage copysets larger than this number.

Wood et al, also propose a hybrid hardware and software directory implementation, called Dir_1SW [24, 49]. Dir_1SW associates two types of state information with each memory block. Dir_1SW handles single copy state transitions in hardware, while it sets the trap bit and then interrupts the processor node for transitions from the `Shared` state. The authors expect most of the simple, one copy, transitions to be performed in hardware, and to handle more complicated ones in software.

More recently, Reinhardt et al. proposed a dynamic user-level shared memory system [40]. Their design includes two components: first, an interface called *Tempest*, which defines mechanisms to handle low overhead messages, bulk data transfer, virtual memory management and fine grain access control to shared data; second, a custom network interface design called *Typhoon* that implements the interfaces defined by Tempest. Although the user-level shared memory approach implements the cache coherence protocols completely in software, its implementation relies on an efficient implementation of the mechanisms in hardware, in particular, of the fine-grain access protection mechanisms.

2.2 Compiler Assisted Solutions

Application-level software-based methods require compiler support to identify accesses to shared variables that might cause cache incoherencies. Veidenbaum presented a compile time algorithm that inserts cache invalidation instructions at parallel loop boundaries [47]. Cytron et al. proposed a mechanism in which cache invalidations were inserted immediately before shared data reads which could access potentially stale data [13]. Later, Veidenbaum's approach was improved by Darnell, Mellor-Crummey, and Kennedy by moving aggregated cache invalidation instructions as close as possible to parallel loop boundaries, at the same time trying not to invalidate data unnecessarily [14]. In the scheme proposed by Lee et

al.[30], the compiler divides programs into segments called *epochs*, and tags variables within an epoch as non-cacheable if they are shared and written, and cacheable otherwise. At the end of each epoch caches are invalidated, and write-back is used to keep main memory updated. Similar techniques are used by the NYU Ultracomputer [17] and the RP3 [8] multiprocessor.

An important advantage of these software-based methods is that they are economical in terms of hardware; only a few instructions used by the compilers to invalidate or flush the cache are needed. Previous research [37, 2] shows that the performance of compiler-based software schemes is comparable to hardware approaches for programs that do not suffer a significant cache hit reduction caused by inaccurate static predictions of memory access conflicts. However, since these schemes must be conservative when determining the interference of shared memory accesses their performance is often not comparable to hardware cache coherence implementations.

2.3 Consistency Models

The different hardware and software cache coherence protocols have been complemented by extensive research on memory consistency models. The memory consistency model provided by the hardware of a shared memory multiprocessor or a software implementation of support for shared memory defines the degree of flexibility allowed in issuing memory accesses. By relaxing memory consistency constraints, techniques such as buffering and pipelining can be used to reduce the latency of memory accesses.

Sequential consistency (SC) [29], the strictest of all memory consistency models, requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. This model allows very little buffering or pipelining of memory accesses.

With relaxed consistency models [16, 3, 20] accesses to ordinary data can be issued and performed in any order, and synchronization points are used as fences that force memory accesses to be globally performed, returning the system to a coherent state. Relaxed memory consistency models therefore allow buffering and pipelining to be exploited at a much higher degree. These methods reduce the total access time to shared data structures by hiding the latency of each memory access.

Release consistency (RC), as defined in [20], distinguishes between two types of synchronization accesses, `Acquire` and `Release`, and categorizes all other accesses as `ordinary`. Under RC, ordinary accesses can be executed in any order with respect to each other; while `Acquires` and `Releases` are used to enforce an ordering across synchronization points.

In software it is possible to take further advantage of the semantics associated with `Acquire` and `Release` accesses. Stores to memory that are required by RC to perform with respect to all other processors on a `Release` operation, can actually be delayed until the next `Acquire` of that lock. This is the main idea behind lazy release consistency (LRC) [26]. Intuitively, the advantage of LRC over RC is that memory updates need to be performed only when locks change owners. Cox et al. showed that the performance of a shared virtual memory system, TreadMarks, implemented on a ATM network of workstations using lazy release consistency can perform close to hardware implementations of directory based shared memory for applications with moderate synchronization and communication demands [12].

In summary, relaxed memory consistency models provide a mechanism to reduce the latency of memory accesses. When implemented in software, these models can be combined

with the semantics of the application, to reduce the the number of operations required to maintain memory consistency.

3 New Software Cache Coherence Algorithms

3.1 The VM-Based Approach

The class of software coherence schemes we investigate requires neither special hardware support, nor compiler support; they can be implemented on shared memory multiprocessors built with off-the-shelf uniprocessors, uniprocessor caches, memory modules, and any interconnections that support memory mapped interfaces. The key feature of these schemes, is that the virtual memory translation hardware on each processor is used to detect shared accesses that could lead to memory incoherencies, and VM page fault handlers execute the appropriate actions to maintain cache coherence.

The idea of VM-based cache coherence stems from previous work on shared virtual memory [32, 9, 7, 12]. It differs from the design of shared virtual memory implementations in that the VM-based cache coherence schemes are destined for architectures that provide a globally shared physical address space, in other words, there is no need to move data between memory modules. The Cray Research T3D exemplifies an architecture that provides globally shared memory without cache coherence, and is therefore well suited for the VM-based mechanisms.

This section describes the basic mechanisms required for the design of the VM-based cache coherence schemes and the algorithms for implementing two memory consistency models, sequential and release consistency.

VM-based cache coherence schemes can be implemented on very simple multiprocessor architectures. They require that the MMU of each processor support the following, commonly available features:

- At least three different protection states associated with a page: *read-only* (READ), *read/write* (WRITE), and *no access* (NIL).
- Physical caches that support selective *write-through* and selective invalidation of cached lines from software.
- Each processor has its own page table, and TLB reloading can be managed from software.

In addition, a *fetch&add* operation at the memory modules is required for efficient implementations of synchronization variables.

The VM-based cache coherence software maintains coherency at the virtual memory address space level. In other words, VM-based cache coherence ensures that accesses to the same virtual address are kept coherent. The granularity at which consistency is enforced is a virtual memory page. When a shared address space is created, the system initializes a page table for each processor. In these tables, the entries (PTEs) for a shared page will always have the same virtual-to-physical address translation. The protection bits, on the other hand, may be different for each processor to reflect the coherency state of the page. VM software will maintain the page tables at run time according to a chosen memory consistency model. The approach is therefore not suitable for applications that run in separate address spaces, yet share physical pages. However, most parallel applications do not operate under that model.

In the design of the VM-based algorithms synchronization variables are non cacheable to avoid the page fault overhead due to locks with high contention. Shared pages will therefore either contain ordinary data, and be cacheable, or synchronization variables, and be non-cacheable. Since most memory management units (MMUs) allow data to be cacheable on a per page basis, this property is used to mark pages that contain synchronization variables as non-cacheable.

By using off-the-shelf parts, state of the art components can be used in building multiprocessors without additional design and fabrication efforts to incorporate cache coherence hardware. This can be a substantial benefit for multiprocessors that try to follow microprocessor technology closely. In addition, VM-based schemes can be implemented on architectures with arbitrary interconnections, thus enabling systems with a large aggregate memory bandwidth. This is an attractive feature, since memory latencies have been increasing in terms of processor cycles times. In general, the VM-based cache coherence schemes trade off simplicity against increased software overheads. The work presented in this paper evaluates this tradeoff.

3.2 VM-Based Sequential Consistency

The following notation will be used for the presentation of the VM-based consistency algorithms:

- P_x denotes processor x.
- $PTE_{x,i}$ corresponds to processor x's page table entry for virtual page i. More specifically it represents the protection bits of the page table entry for virtual page i.
- $TLB_{x,i}$ corresponds to processor x's TLB entry for virtual page i. Again, this notation refers specifically to the protection bits in the specified TLB entry; and

As defined in Section 2.3, Sequential Consistency (SC) requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. In order to support sequential consistency, the algorithm for VM-based SC allows one exclusive writer on a page, or, in absence of a writer, multiple readers. The VM-based algorithm of SC is shown in Figure 1. First, we will present the VM-based cache coherence algorithms, assuming that write-through is used to update the memory modules for shared data. Later in this paper, we will also discuss and evaluate the implementation of the VM-based schemes with write-back caches. Writing shared data through to the memory modules alleviates the need for flushing individual lines from the cache, which simplifies the algorithms presented below.

Figure 2 illustrates this algorithm with a transition diagram for the state of a page. The diagram shows six possible transitions of a page's protection setting, along with the actions taken on each transition. In the diagram, a *remote page fault* means a page fault on a different processor.

The cost of accessing each processor's page table entry for page i is proportional to the number of processors in the system. Although the cost of finding one page table entry is only a few instructions and one memory load, the cost of accessing the page table entries of all processors can be significant. In the implementation of the VM-based coherence schemes, one status word is used for each shared page to keep track of which processors have access to the page. The status word is used either as a bit-vector to indicate the copyset, that is, all readers of the page, or to encode the processor number of the current writer of the page. The protection bits in the page table entries of each processor are kept consistent with the status words. For each shared address space the system will have a separate set

of such shared page status words. The table containing these shared page status words is non cacheable.

3.3 Solution to the False Sharing Problem

The main disadvantage of the VM-based SC implementation is that it can suffer from high levels of false sharing. For systems where cache coherence is maintained in hardware and cache lines are 4 to 64 bytes long, cache misses due to false sharing can be as high as 40% [18]. For the VM-based implementation of SC, the cache coherence unit is a page, which is much larger than a regular cache line, and therefore much more likely to contain several different data items that are accessed by different processors simultaneously.

One solution to this problem is to conduct a careful analysis of the application's data dependencies and allocate data manipulated mostly by one process on pages with no data for other processes. However, the analysis of data dependencies is difficult, in particular for dynamically allocated data structures[23].

Another solution, is to allow processors to access different data items in the same page simultaneously, and use synchronization events to enforce that changes performed by each processor get merged consistently. Synchronization events are therefore used to keep memory in a coherent state. Relaxed consistency models are based exactly on this idea. Several researchers have proven that for applications, whose accesses to a shared data item are properly guarded by synchronization primitives, the execution is guaranteed to be correct under a relaxed memory consistency model [1]. Although relaxed consistency models were initially designed to hide long memory latencies, for the VM-based cache coherence schemes they can also avoid page faults otherwise caused by non-conflicting accesses to different cache lines in a page.

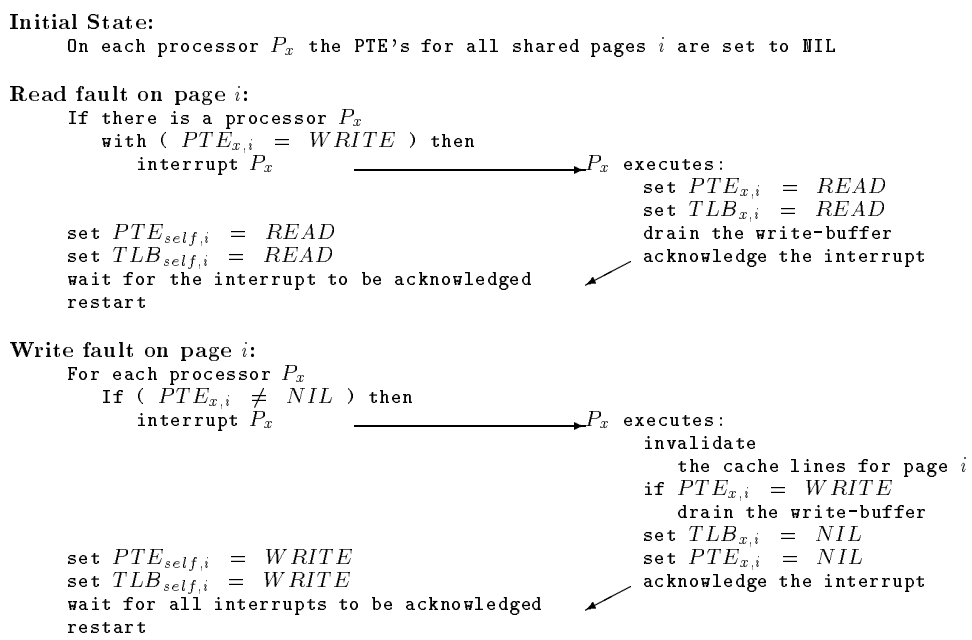

Figure 1: VM-based algorithm for SC

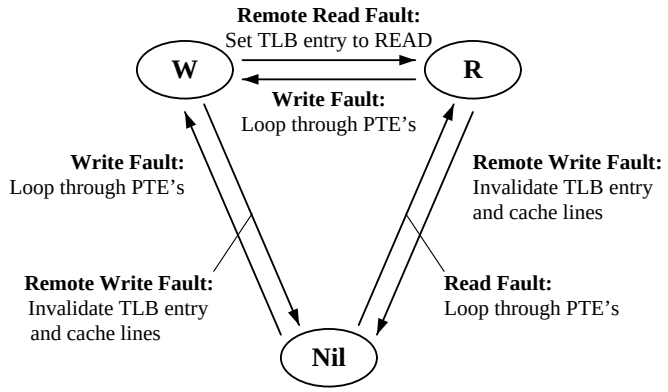

Figure 2: Transition diagram for sequential consistency.

3.4 VM-Based Release Consistency

The algorithm for VM-based release consistency combines techniques from both the hardware implementation of RC and LRC, the software implementation proposed for distributed memory machines. VM-based RC is closer to RC in that it forces all stores to shared data to perform with respect to the memory modules before a `Release` operation completes. However, VM-based RC is similar to LRC, in that it postpones all cache line invalidations until the next `Acquire`.

Synchronization variables are kept sequentially consistent by ensuring that memory accesses do not complete until performed at the memory modules. In architectures that include a write-buffer, a processor will have to stall until its write-buffer drains, following a store to a synchronization variable. Notice that with a write-through cache, upon completion of a `Release`, all stores issued before the `Release` are guaranteed to have been performed at the memory modules as well.

In VM-RC, page faults are used only to record when a processor gets access (READ or WRITE) to a shared page. This allows the system to keep track of shared pages that are being shared and updated and might become incoherent.

On `acquires` a processor needs to invalidate the cache lines of shared pages to which it has access and that were also updated by any processor that previously held the lock. For a processor to identify these pages the algorithm maintains a globally shared list of pages called the `WeakList`. A page is added to the list when it becomes shared and writable, and is removed from the list when all processors have relinquished (READ or WRITE) access to it. On an `Acquire`, a processor checks all pages in the `WeakList`, and performs the appropriate invalidations on all pages that it has access to. The `WeakList` is kept in shared memory, but cannot be cached. Two words for each page are enought to maintain the list: one to keep the page number for the next page on the list, and another one $weak_i$, a counter, to keep track of the number of processors that have access to page i at any given time.

The algorithm for the VM-based implementation of RC is shown in Figure 3. Notice, that in contrast to the algorithm for VM-based SC, the algorithm for VM-based RC does not require any processor synchronization. Enough synchronization is provided by the applications for the algorithm to execute locally.

```
Initial State:
    On each processor P_x the PTE's for all shared pages i are set to NIL
    For all shared pages weak_i = 0
    WeakList = NULL

Read fault on page i:
    Increment weak_i
    If there is a processors P_x
       with ( PTE_{x,i} = WRITE ) then
          make sure page i is on the WeakList
    set PTE_{self,i} = READ
    set TLB_{self,i} = READ
    restart

Write fault on page i:
    If ( PTE_{self,i} = NIL ) then
       increment weak_i
    If ( weak_i > 1 ) then
       make sure page i is on the WeakList
    set PTE_{self,i} = WRITE
    set TLB_{self,i} = WRITE
    restart

On Acquire of lock X: using write-through
    For all pages i ∈ WeakList
    If ( PTE_{self,i} ≠ NIL ) then
       invalidate all cache lines for page i
       set PTE_{self,i} = NIL
       set TLB_{self,i} = NIL
       decrement weak_i
       If (weak_i = 0) then
          remove page i from the WeakList
    endif

On Release of lock X:
    wait for the write-buffer to drain
```

Figure 3: VM-based algorithm for RC

4 Performance Evaluation

4.1 Trace Driven Simulation

Trace-driven simulations were used to evaluate the proposed VM-based cache coherence methods. In order to compare the VM-based methods with a hardware-based cache coherence mechanism, the test-suite is based on application programs implemented for shared-memory multiprocessors with per-processor caches and hardware based cache coherence protocols. The applications were modified only to allocate synchronization variables on pages separate from other shared data. The application's algorithms, memory allocation strategies, and data placement remain the same. Comparison with a known hardware-based cache coherence mechanism will allow us to evaluate the effectiveness and performance of the VM-based coherence schemes.

The architecture that we simulate initially consists of 4 uniprocessor nodes connected to memory via a split transaction bus. The main reason for performing our comparisons with a bus based system, rather than another kind of network, is to compare the VM-based methods with a snoopy cache scheme, which is the most commonly implemented hardware-based cache coherence approach. Later in the paper we also present how the VM-based schemes perform on crossbar interconnections. Table I shows a listing of all the architectural parameters used in the simulations. Although we chose to compare the VM-based schemes with only the bus-based snoopy cache approaches, the work by Kontothanassis and Scott demonstrates how the VM-based schemes compare to directory-based approaches [27].

The instruction set of the simulated processors resembles the MIPS R3000. For the initial evaluations each processor node has a 20K direct mapped data cache on chip. Writes are buffered in an 8 entry write-buffer. Depending on the cache coherence protocol, loads may bypass stores queued in the write-buffer. A load operation to an address in the write-buffer will not generate a bus transaction, but instead return the value stored in the buffer. Adding a word to the write-buffer requires one processor cycle, and write-around is used

	Base value	Evaluated Alternatives
Processor		
CPU cache	20 Kbytes direct mapped	64 Kbytes direct mapped
Block size	32 bytes	
Page size	4 Kbytes	1K, 16K, 64K
Write buffer	8 word (4 byte) entries	
Local write	1 cycle (write buffer access)	
Interconnection		
Topology	Bus	Crossbar
Data lines	32 bits	
Arbitration	1 cycle	2 cycles
Address latching	2 cycles	4 cycles
Data send	2 cycles	4 cycles
Memory modules		
Memory setup	10 cycles	15 cycles
Page mode setup	2 cycles	4 cycles

Table I: Simulation Parameters

on cache write misses. The memory unit consists of four interleaved memory modules and takes advantage of page mode; the interleaving occurs on a per cache line basis.

The bus has separate address and data lines. The costs for the phases in a bus transaction are shown in Table I. Since the cache line size is 32 bytes, the round-trip latency for a cache line fill in the absence of bus contention is 29 cycles. The scheme used for bus arbitration is very simple; load requests have priority over write-buffer requests, and requests of the same type are served on a first-come-first-served basis.

Every machine instruction is assumed to take one cycle, which sets an upper bound on the stress exercised on the memory system. Although the instruction cache is not simulated, instruction misses are artificially inserted with probability of 0.002 at each instruction.

To validate the TLB-miss overhead simulated for the VM-based cache coherence schemes, we measured the average time it takes to handle a TLB-protection miss on a DECstation 5000/200 running Ultrix Version 4.3. Although the MIPS R3000 processor provides mechanisms to handle TLB replacement very fast, the VM-based mechanisms rely on the protection faults, TLB Invalid and TLB Modified, which are serviced by the common exception handler. We measured the overhead of this exception handler, optimized to quickly identify protection faults caused by cache coherence management and to make better use of register allocation. An extensive discussion of these measurements can be found in [39]. In summary the optimized exception handler required an average of 226 cycles to execute. This number is exactly in the rage of the page fault times (182–285) reported by the simulations of the VM-RC algorithm.

The application suite includes a Gaussian elimination program, matrix multiply, and four programs from the SPLASH parallel applications: LocusRoute, WATER, MP3D, and Cholesky [48, 43]. All applications are coded in C, and use the *parmacs* macros from Argonne National Laboratory [34] to implement parallel operations such as: creation of cooperating processes, allocation of shared memory, and usage of lock and barrier synchronization variables.

4.2 Results

This section presents a comparative analysis of the performance achieved by the following cache consistency models:

- **Snoop** — The Illinois, invalidation-based, snoopy cache protocol [38].[1]
- **NO** — No caching of shared data.
- **VM-SC** — VM-based sequential consistency as described in Section 3.2.
- **VM-RC** — VM-based release consistency as described in Section 3.4.

All models were simulated using the same architectural parameters.

Normalized execution time is used to characterize the performance of the VM-based coherence schemes with respect to the hardware snoopy cache approach. In each graph, the execution time for the snoopy scheme represents 100%, and the execution time of all other schemes is normalized to it. In our experiments, execution time is measured as the number of cycles required for all processors to simulate the whole trace for a given cache coherence protocol.

To better understand the results, the graphs also show a breakdown of the execution time for all applications with each of the evaluated cache coherence schemes. The breakdown is based on the average time one processor spends in each section. Four major regions can be identified in the breakdown: cycles a processor is idle due to network contention, the rest of the time spend waiting for memory accesses to complete, cycles due to synchronization or coherency overhead, and finally cycles a processor is busy with application code (Figure 4).

4.2.1 NO versus Others

The most economical and the simplest method of cache coherence is implemented by not caching shared variables, i.e., the no-caching scheme (NO). However, as expected, it has poor performance for most applications. Figure 5 shows the performance of Snoop, NO and VM-SC for each application. We can see that there can be a significant performance gain from using VM-SC, rather than no-caching. This is true for four of the applications we evaluated.

There are two reasons for MP3D's poor behavior: it has low locality of reference, and a high degree of interference caused by processors accessing the same shared memory locations. The low locality is a result of statically assigning a set of molecules to each processor.

[1] In Snoop the caches are write-back. It would be unfair to force the Illinois protocol to suffer contention due to write-through when it does not need it.

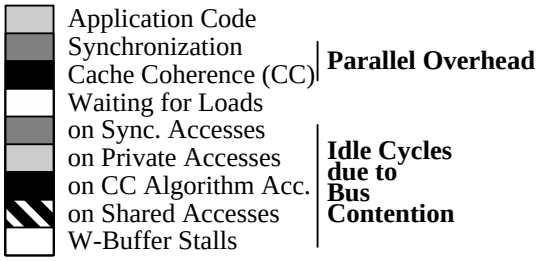

Figure 4: Legend used in graphs.

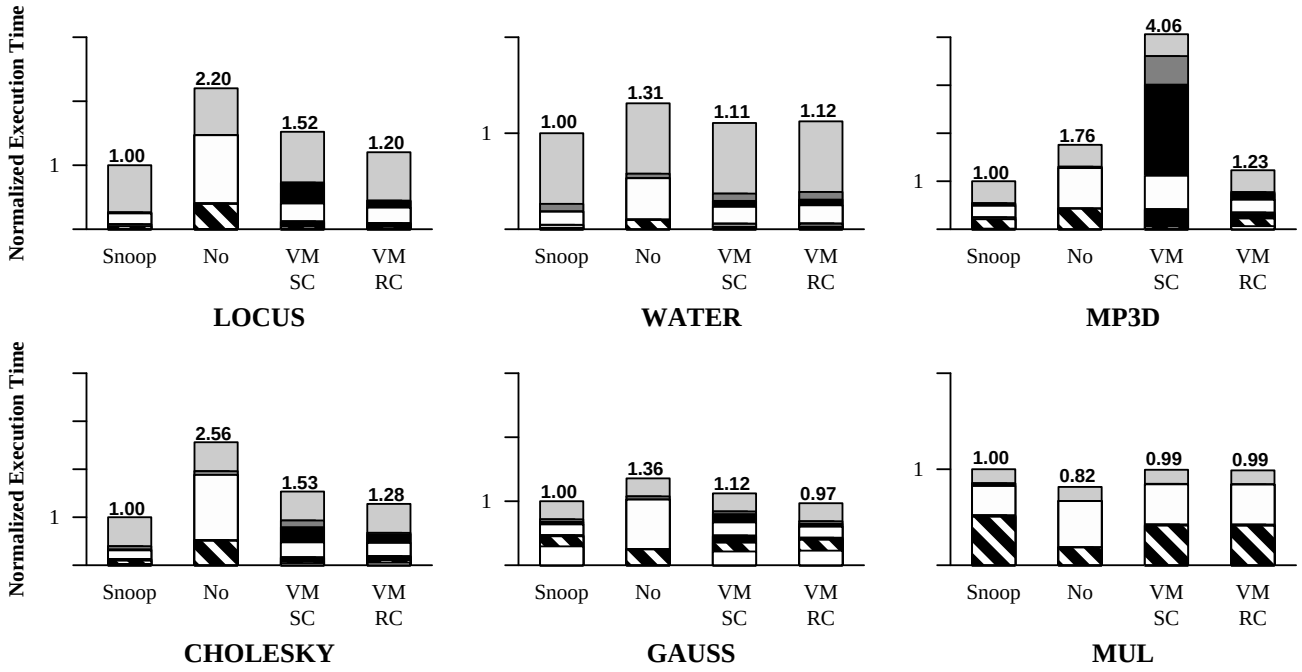

Figure 5: Performance of the VM-based schemes.

Over time the molecules move into different space cells, and data interference occurs when molecules managed by different processors move into the same space cells and need to update the cell's data structures. In addition the granularity of our coherence unit, 4096 bytes, adds to this factor by introducing high degrees of false sharing. Page faults occur at an average rate of 1 in every 13 accesses for the sequential consistency mechanism, of which 65% are false misses.

MUL performs best with no caching because the algorithm was not blocked to fit 20K caches, and therefore only a little over 50% of all shared accesses hit in the cache. Cache refills require 32 bytes to be loaded from the memory modules, while under NO the memory accesses to shared variables only require 8 bytes to be transferred from memory to the processor. This means that the caching mechanisms, Snoop and VM-SC, were actually transferring twice as much data over the bus and making the bus a serious bottleneck. In addition NO exploits locality of private data structures much better, since the cache is not polluted with shared data.

In GAUSS, during one pivoting phase, each processor's cache fills after the 16th row has been eliminated. At that point, every fourth access (the first access in a cache line), will cause a cache replacement of a dirty line. This is why NO does not perform much worse than the caching schemes for GAUSS.

For GAUSS and MUL the reduced performance benefits from VM-SC are due to the cache size. The performance gains of the VM-based SC approach over NO are considerable in all other cases, ranging from 16% to 40%. The application with the lowest performance gain is WATER, because it has the lowest number of memory accesses to shared data. With an increasing number of references to shared data, caching becomes more critical. For example, with 40% of the memory accesses in LocusRoute being to shared data, the performance gain is 30%. Furthermore, as shown in Section 5.3, longer memory latencies also make the caching schemes more cost-effective.

4.2.2 RC versus SC

Figure 5 also shows the effect of VM-based RC on the performance of all applications. In general, RC performs best among the VM-based protocols. The performance gain over SC depends on the degree of contention for shared pages. For MP3D, 8% of all shared accesses are to pages already accessed by other processors, or reads from a page written by another processor, hence the large improvement in performance obtained by relaxing the memory consistency constraints. MP3D performs 70% better with VM-RC, than with VM-SC. All other applications have less contention for shared pages. Less than 1.5% of shared accesses are to pages that cause contention because of coherence. For these applications, the performance improvement is less than 22%.

For MP3D the overhead to keep coherency decreases from 55% of the total execution time in VM-SC, to only 10% in VM-RC. The page fault ratio is reduced from 1 in every 13, to 1 in every 164 shared accesses. This accounts for most of the improvement. In addition, because the memory consistency requirements are relaxed, the cache miss ratio also decreases from 9.8 in VM-SC to 6.9 in VM-RC.

For all applications, the overhead due to coherency faults decreases in VM-RC because of the relaxed memory consistency restrictions, and accounts for most of the improvement of each application. In absolute terms, the reduction in coherence overhead is 67% for LOCUS, none for WATER, 90% for MP3D, 36% for CHOLESKY, 80% for GAUSS, and none for MUL.

WATER actually performs slightly better with VM-SC. Although the page fault ratio is reduced in VM-RC by about a factor of two, the reduction is not sufficient to amortize the increased cost of synchronization events.

4.2.3 RC versus Snoop

The performance of VM-RC, as compared to Snoop, varies depending on the application. GAUSS performs best, within three percentage points of the snoopy cache approach. CHOLESKY, however, requires 30% more time to execute under the VM-RC approach than under Snoop.

The execution time breakdown for each application shows two reasons for the difference in performance between VM-RC and Snoop: (1) the overhead due to the VM-based implementation of RC, (2) the time a processor stalls on a full write-buffer, and (3) the time a processors spends waiting for memory loads to complete.

The overhead of the VM-based implementation of RC is a function of the number of pages simultaneously shared and updated by the processors and the frequency of synchronization events that require coherency actions. In other words, it depends on the length of the `WeakList` when a barrier is reached or a lock's ownership is transferred from one processor to another. This information, in addition to the time idle due to write-buffer stalls, is summarized in Table II. For CHOLESKY, which has the worst performance compared to the snoopy approach, the overhead due to the RC algorithm accounts for 27% of its execution time, and write-buffer stalls consume 4.2% of the execution time. CHOLESKY therefore spends 30% of its execution time in the VM-based implementation of RC. Although GAUSS' total overhead due to VM-RC is also around 30%, its behavior is different from CHOLESKY; only about 6% of its execution time is in RC overhead, and 24% is due to write-buffer stalls. However, GAUSS' performance is not 30% worse than Snoop's, because most of the time GAUSS spends on write-buffer stalls in VM-RC is also required

in Snoop to write back dirty cache lines. For MUL, on the other hand, the total overhead is very small, less than 0.1%.

Application	Average WeakList Length	Coherency Synch. Events	**Algorithm Overhead** per processor (% of total execution time)			
			Instructions	Memory Accesses	W-buffer Stalls	Total
LocusRoute	29.3	407	11.2	8.2	0.5	19.9
WATER	3.2	2596	5.3	5.3	0.2	10.8
MP3D	15.0	35	10.5	10.8	4.1	25.4
CHOLESKY	3.8	3819.7	13.9	12.6	4.2	30.7
MUL	2.0	2	0.05	0.02	0.0	0.07
GAUSS	1.0	320	4.6	1.1	23.7	29.4

Table II: VM-RC Overhead.

As mentioned previously, even though GAUSS' overhead is so high, it actually performs better than Snoop. This occurs because most of the overhead of GAUSS under VM-RC is due to write-buffer stalls, which is also present in Snoop when dirty cache lines get replaced. Furthermore, since LOADS do not bypass STORES in the write-buffer of our simulation of Snoop, GAUSS frequent updates to shared data structures affect Snoop more than VM-RC. With larger caches, applications that update shared data frequently like GAUSS will affect VM-RC more strongly. With Snoop cache lines will not be replaced as often, hence generating less write-back traffic, while the write-through traffic of VM-RC will remain the same. In addition, modern bus-based snoopying schemes implement some sort of relaxed memory consistency, thus allowing LOADS to bypass STORES and removing the benefit GAUSS sees in our simulations. Nevertheless, as discussed in Section 5.2, techniques such as coalescing write-buffers can nearly eliminate the performance penalty caused by write-through policies, thus removing the differences in performance that are artifacts of our simulation parameters.

The cache miss ratio under Snoop and VM-RC is very similar for all applications, therefore the number of memory accesses due to shared and private accesses that require a bus transaction is very close. The total memory access time in VM-RC is affected mostly by memory accesses issued by the algorithm and due to write-throughs.

Relaxing the memory consistency models, and hence synchronizing the processors in the system only when the application requires to do so, allows the VM-based RC cache coherence scheme to perform very well.

4.2.4 Crossbar Interconnections

As we observed above and discussed in Section 3.1, the load on the memory bus greatly affects the performance of the VM-based RC scheme. One of the main advantages of the VM-based schemes is their independence of specific interconnection topologies. Any interconnection network with intrinsically lower contention than a bus, like a crossbar or mesh, can be used to connect processors to memory modules, hence providing higher aggregate memory bandwidth.

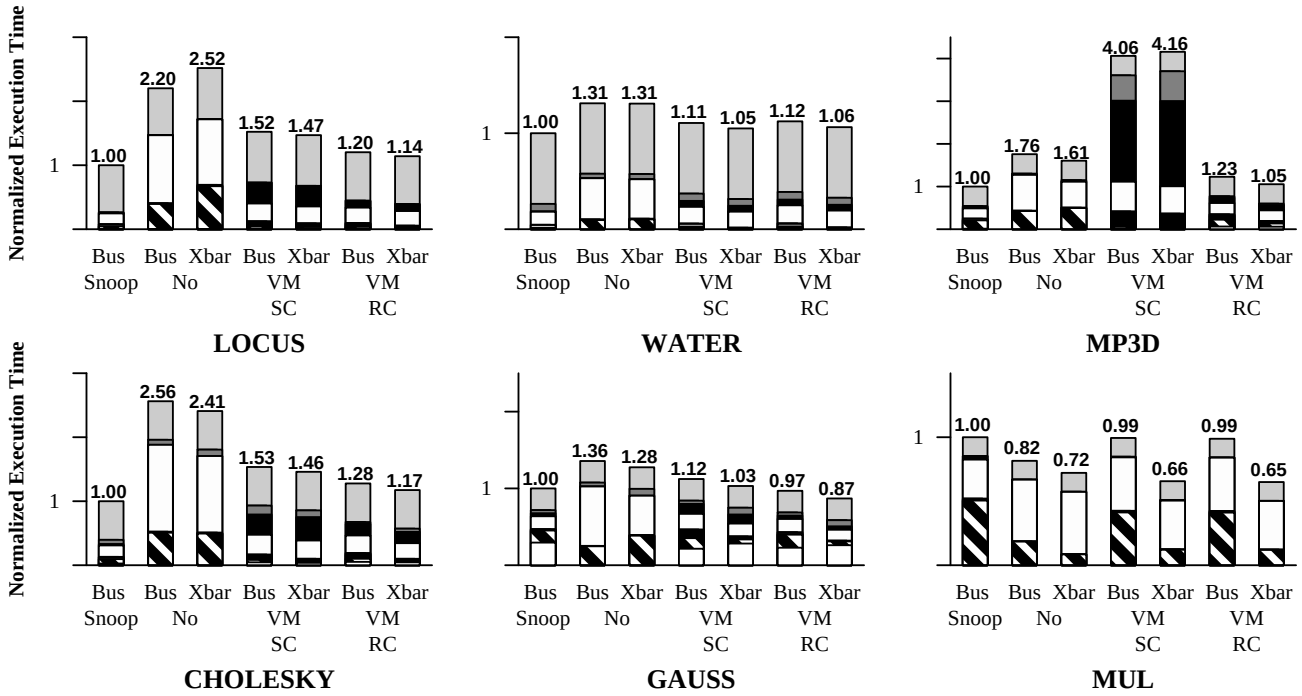

Figure 6: Performance with crossbar interconnections.

Figure 6 shows the performance of the cache coherence protocols, simulating a crossbar network for all models with the exception of Snoop. The latencies on the crossbar are the same as for the simulated bus.

The graph shows that all applications benefit from architectures built with lower contention interconnects. The improvements vary depending on the ratio of memory access time to computation, and the amount of bus contention caused when network memory accesses are frequent. For MUL, which spends close to 85% of its execution waiting for memory accesses to complete on a bus based architecture, improves its performance by 34% on a crossbar architecture with the VM-RC cache coherence scheme. Multiple paths to the memory modules reduce contention on the network by 70%. For applications, such as WATER, that spend much less time waiting for memory accesses, the improvement can be as small as 5%.

One would have expected the idle time due to write-buffer stalls in GAUSS to be reduced substantially as well. However, when moving to the crossbar interconnection, in our simulations the memory interleaving changed to be page based, rather than, cache line based. Memory accesses to the same pages are hereby serialized. In GAUSS, shared memory updates are sequential, hence generating cache write-throughs to sequential addresses that do not benefit from the multiple paths to the memory modules. Write-buffer stalls should be reduced, if interleaving is added to the memory modules at each processor.

The no-caching approach also benefits from low contention interconnects, but not anywhere near to the 75% reduction in contention that one would hope for with four paths to the memory modules. The reason is similar to why write-throughs do not scale either; accesses to the same page are serialized. If in addition several processors access pages on the same module, the memory setup time cannot be parallelized.

However, the VM-based RC schemes perform at least 1.1 and up to 2 times better than NO for all applications, and within 17% of the hardware snoopy cache coherence implementation. For applications, such as GAUSS and MUL, that have high cache miss ratios, and therefore require high bandwidth from the interconnection, having multiple paths to the memory modules improves performance significantly.

5 Impact of Architectural Parameters

5.1 Coherence Unit

The cache coherence unit for the VM-based cache coherence algorithms is a virtual memory page. The size of the virtual memory page should affect the algorithms for SC and RC differently. SC benefits from smaller pages, since smaller pages decrease the degree of false sharing. For applications with minimal amounts of sharing, larger pages would improve the performance of SC, but since these applications are rare, it is likely that large pages will make the overhead of SC unbearable.

Simulation results for VM-SC executing with large pages show that for two of the applications, WATER, and MUL, the effect of the page size is not significant since most accesses to shared data cause very little contention. However, for the other four applications, LOCUS, MP3D, CHOLESKY, and GAUSS, the page size has a much more noticeable effect on performance. For these applications the VM-SC protocol performs best with the smallest page size. This is because smaller page sizes reduce false sharing. As the page size increases the number of page faults caused by false sharing increases as well. For LOCUS this increase is the smallest, from .8% faults with 1K pages to 1.8% with 16K pages. MP3D shows the largest page fault increase: from 7% faults with 1K pages to 17% with 16K pages.

The effect of page size on the performance of VM-based RC is not as straightforward. Figure 7 exhibits the effect of changing the page size for VM-RC from 4 Kbyte to 64 Kbytes. Larger page size ranges were studied for VM-RC than for VM-SC, since false sharing does not have the substantial effect it does on the VM-SC scheme.

From the results one concludes that in general the effect of the page size on VM-based RC is minor. The largest performance difference occurs for MP3D, where 64K pages improve the performance by 10%. MP3D's performance improvement is not surprising, since most of its pages become shared and updated during one phase of its execution. With larger pages the average number of pages on the `WeakList` is reduced from 15 with 4K pages to 3 with 64K pages. Because almost all pages become shared and updated the number of page faults is also reduced with larger pages, on average from 1 fault in every 160 shared data accesses with 4K pages, to 1 fault in every 900 accesses with 64K pages. For LOCUS increasing the page size has a similar effect as for MP3D. The performance improvement for LOCUS is 7%, which is smaller than for MP3D.

For WATER, CHOLESKY, and GAUSS, however, increasing the page size, slightly deteriorates performance. In fact the number of page faults increases because of false sharing. This causes more cache invalidations and therefore more memory traffic. For CHOLESKY the cache miss ratio increases from 4.33% with 4K pages to 6.6% with 64K pages, which causes the cycle count due to memory traffic to increase by 24%. For WATER and GAUSS the effect is similar, but less noticeable.

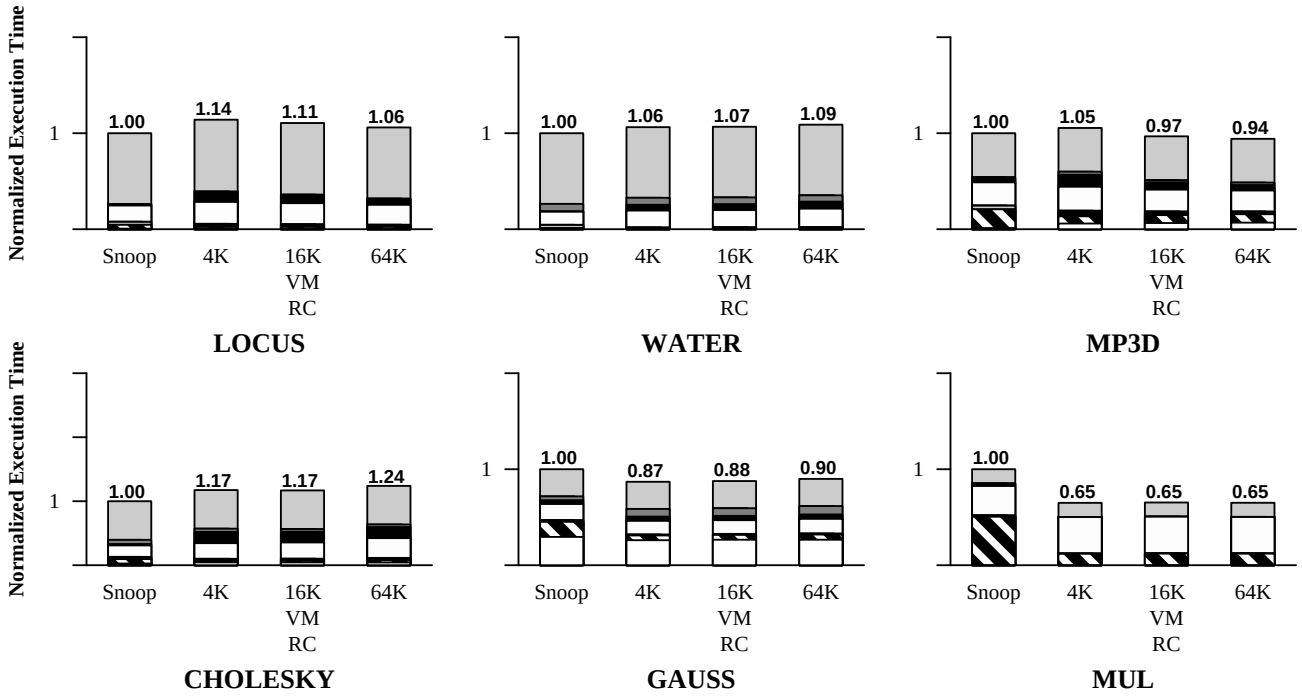

Figure 7: Effect of page size on VM-RC performance.

5.2 Write-Back Caches

When compared to write-through caches, write-back caches can reduce the load on the interconnection network between processors and memory modules, because memory requests are not issued for every store operation. This is particularly important on networks that are already heavily loaded. To implement the VM-based cache coherence schemes with write-back for shared data requires changing the algorithms. Therefore, the performance tradeoff for the VM-based schemes resides in the amount of network contention that is avoided versus a potential increase in algorithm overhead.

When an `Acquire` is executed in the VM-based RC algorithm with write-through caches, invalidating all cache lines for pages on the `WeakList` guarantees access to all updates generated by processors that previously held the lock. This occurs because the most recent values will be loaded from the memory modules during cache refill operations. However, on an architecture with write-back caches, it is not sufficient to invalidate all cache lines for pages on the list, since updated values may not have reached the memory modules, but may remain in the cache of the processor who previously owned the lock. It is therefore necessary for the previous owner to flush updates to shared data structures back to memory, before the current `Acquire` can perform.

To implement the VM-based RC algorithm for write-back caches, an additional status word for each lock, $lastowner_L$, is used to identify which processor last held the lock. The remaining changes in the algorithm occur on `Acquires` and `Releases` as shown in Figure 8. Notice that only one processor will request that the previous owner flush its cache, since the flush request is only issued after a processor is holding the lock. An `UpdateList` on each processor keeps track of all shared pages that have been updated by that processor since it last serviced a flush request.

```
On Acquire of lock X:
     Interrupt lastowner_X                    P_lastowner_X executes:
     For all pages i ∈ WeakList                   For all pages i ∈ UpdateList_self
        If (PTE_self,i ≠ NIL) then                   If (PTE_lastowner_X,i = WRITE) then
          invalidate all cache lines                   flush all cache lines for page i
            for page i                             drain the write-buffer
          set PTE_self,i = NIL                     acknowledge the flush
          set TLB_self,i = NIL
          decrement weak_i
          If (weak_i = 0) then
            remove page i
            from the WeakList
        endif
     wait for the acknowledgement
     set lastowner_X = P_self
On Release of lock X:
     do nothing other than issuing the Release.
```

Figure 8: VM-RC using write-back for shared data

Figure 9 shows that using write-back for shared data does not improve the performance of the VM-RC algorithms, on the contrary the performance degrades. LOCUS, WATER and CHOLESKY are the applications affected most negatively by the change to write-back for shared data. This result was somewhat unexpected, since it is common knowledge that write-back policies reduce contention on the memory interconnect. In particular, we had expected that GAUSS may benefit from using write-back caches.

The overhead of VM-based RC with write-back caches increases because of the additional synchronization required to flush the caches on lock ownership transitions. Although network contention for shared cache reloads is reduced with write-back caches, the reduction is minimal. The best result is shown by MP3D, where idle cycles due to contention on cache reloads of shared data diminish from 9% to 4%. The purpose of using write-back caches to reduce the impact of write-buffer stalls, is not achieved. On a flush request all stores to the write-buffer occur at once, and the actual number write-buffer stalls increases for most applications. We did not realize this when designing the algorithms.

Although, in general, the VM-based cache coherence schemes perform much better for architectures which support write-through for shared data structures, VM-RC with write-back caches is 10–40% faster than not-caching for all applications except WATER. WATER does not access shared data frequently enough to pay the overhead of VM-RC with write-back over not-caching. With an effective mechanism to eliminate the stalls, such as using a coalescing write-buffer, suggested by Jouppi [25], the performance of the write-through based VM-RC could be further improved. Kontothanassis and Scott show that this is in fact the case [27]. In Petersen's thesis [39] we also show how simple source-code annotations can significantly reduce the overhead due to shared updates.

5.3 Memory Latency

Accesses to the physical memory modules by all VM-based cache coherence algorithms fall into three categories: cache write-throughs, cache reloads, and accesses to the algorithm's data structures. For cache reloads and accesses to the algorithm's data structures, longer memory latencies directly imply longer execution times. Frequent stores to shared data can fill the write-buffer and then cause the processor to stall on further stores. In addition, the

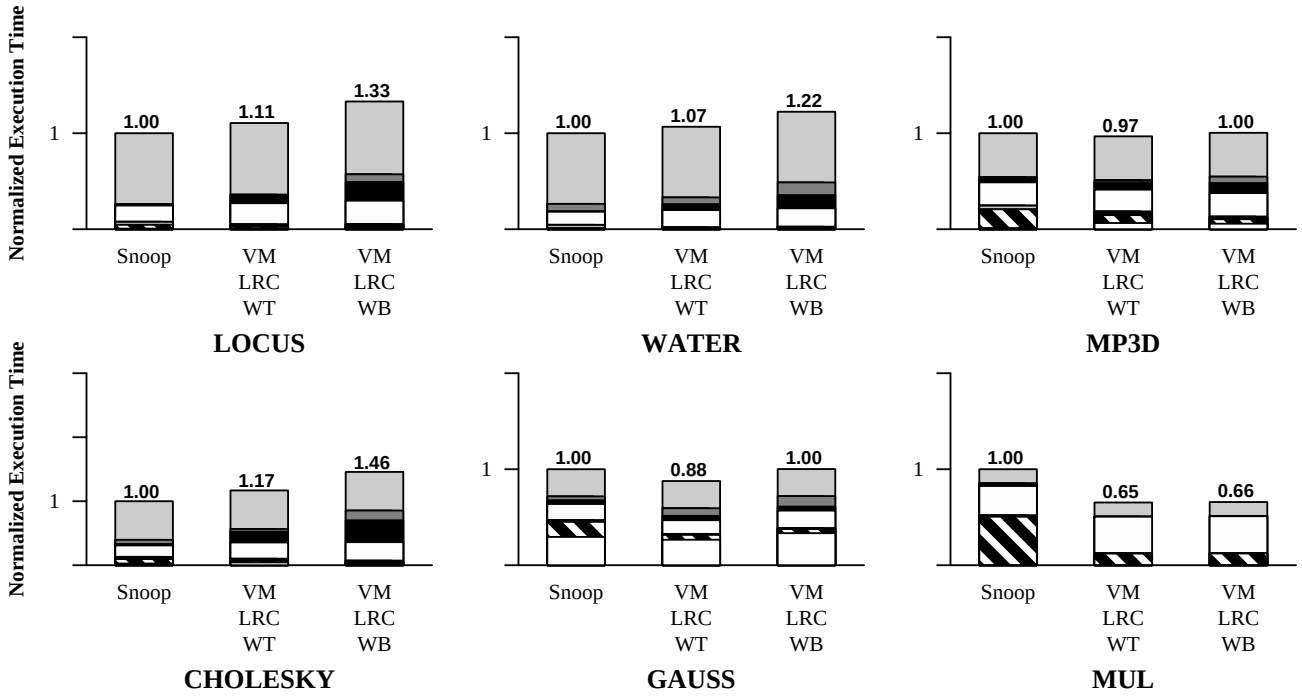

Figure 9: VM-RC performance with write-back caches and UpdateLists.

overhead of cache reloads and algorithm data accesses can also be increased slightly due to inflight write-buffer requests.

With larger memory latencies the performance of all algorithms would be affected mostly by contention for the interconnection network. Longer latencies mean that the probability of a memory access being issued when the interconnection is already busy increases. The advantage of the VM-based algorithms with respect to the contention problem, is that they are not bound to bus topologies. Consequently, the negative effect of contention due to longer memory latencies on the performance of the VM-based algorithms can be reduced by using architectures with multi-path interconnection topologies, such as crossbars and meshes.

The performance gain of moving from NO to the VM-based schemes is very sensitive to the memory latency cycle times. As the gap between microprocessor speed and DRAM speed continues to increase, we expect the performance gain from changing to the VM-based schemes, instead of NO, to increase as well.

To evaluate the effect of memory latency, we simulate an architecture with lower bandwidth interconnections and slower memory modules. The increased costs in a memory access transaction are as follow: 2 cycles for arbitration, 4 cycles for address latching, 4 cycles to send 4 bytes of data, 15 cycles latency at the memory modules for the first access to a page, and 4 cycles for subsequent accesses to the same page. A cache line fill in the absence of contention takes 53 processor cycles, almost twice the time of a cache refill with the original parameters. As expected, the performance of all applications deteriorates with longer memory latencies.

Figure 10 shows the relative performance of all applications. The graph does not allow to compare the absolute performance of the shorter and longer memory latency results, but is used for the relative comparison of the schemes in the two cases. In the graph, all results

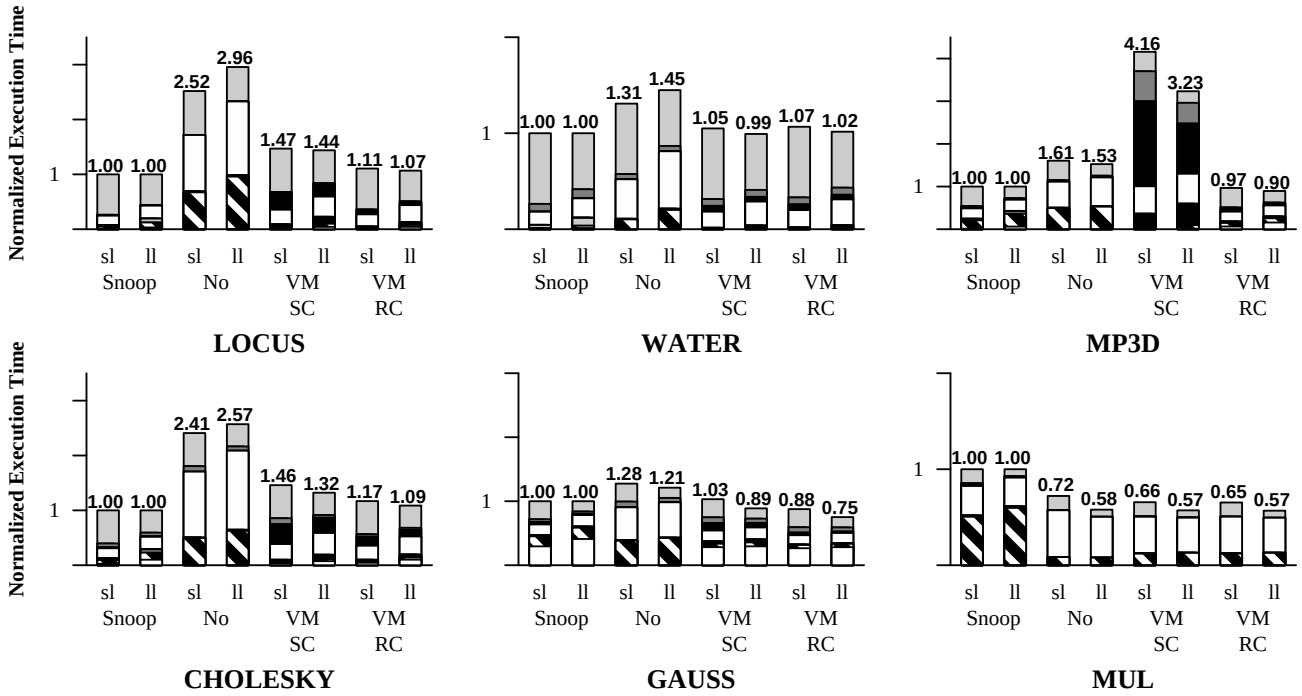

Figure 10: Relative performance with varying memory latencies.

for the short latency interconnection are normalized to Snoop with *sl*, while the results for the high latency interconnection are normalized to Snoop with *ll*. This figure demonstrates the pattern for the performance of the VM-based schemes with respect to Snoop and NO.

Relative to the Snoop approach on the corresponding architecture, NO with long memory latencies takes almost up to 3 times longer to execute, while with faster interconnects the largest ratio is 2.5 (for LOCUS).

In relative terms, the performance of VM-RC with respect to Snoop improves; VM-RC performs closer to Snoop on architectures with longer memory latencies. This occurs because longer memory latencies have a larger impact on bus based architectures. On the average, network contention under Snoop triples with the simulated lower network bandwidth, while for VM-RC it increases by only little more than double. In addition, the contention on the bus network is consistently at least twice, sometimes three times, the one for the crossbar. Because the increase in bus contention is so significant, and because the effect of memory accesses on the execution time of the applications is larger with lower bandwidth interconnects, the relative performance of the VM-based schemes improves when compared with Snoop, the hardware scheme. This means that the VM-based schemes are even more cost effective on architectures with longer memory latencies because they can be implemented on multi-path interconnections.

There is a slight caveat to this conclusion; when memory latencies increase mostly due to memory module set up time, and not because of network bandwidth, the relative performance of the VM-based schemes when compared to Snoop will depend on contention at the memory modules. One approach to decrease contention on the memory modules it to provide a large number of them, which in turn means that several outstanding memory accesses will eventually contend for the interconnection network, and non-bus interconnection topologies will improve overall performance. On the other hand, contention at the memory

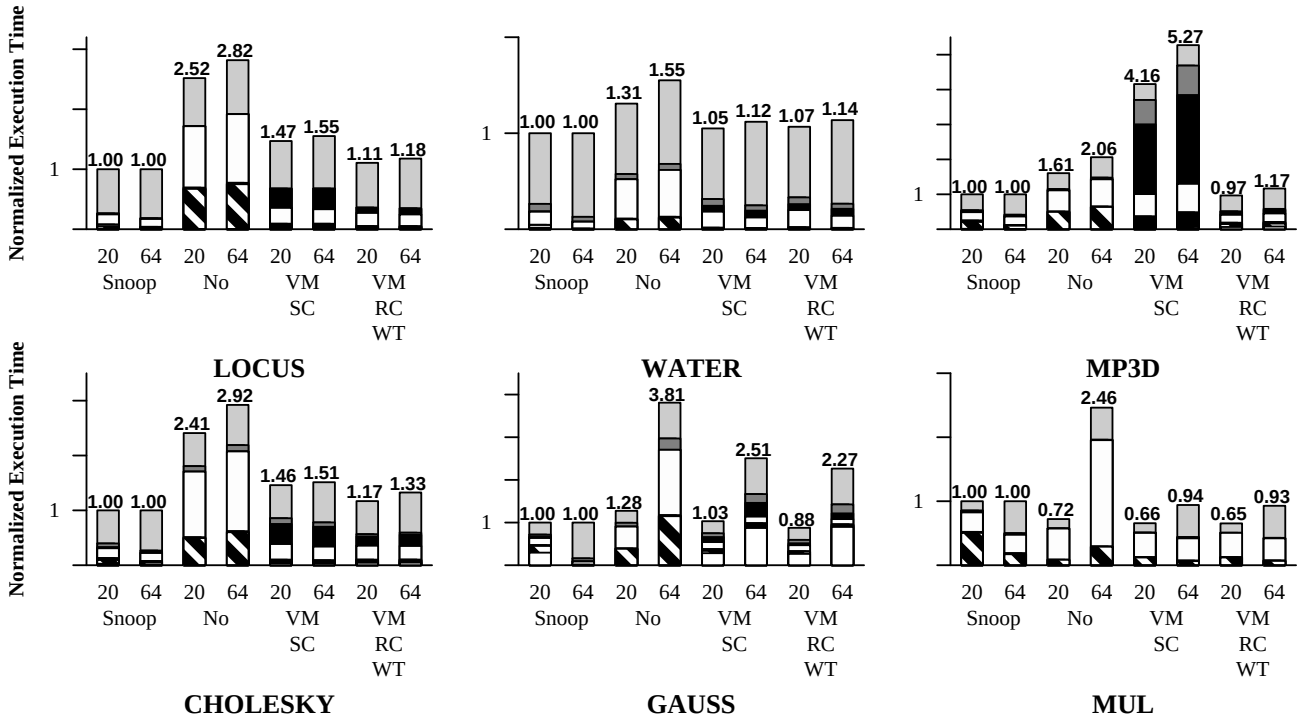

Figure 11: Relative performance with varying data cache sizes.

modules can also be reduced by cache-to-cache communication as provided by most snoopy caches. Snooping mechanisms will therefore benefit applications with high data locality, while the VM-based systems will benefit applications with less data locality.

5.4 Cache Size

Since simulations are considerably slower than the real architectures being simulated, only small data sets can be used to evaluate the system. Small data sets cause few capacity cache misses and therefore do not portray the behavior of applications with realistic data sets accurately. However, evaluating the effect of $64K$ caches will give some insight of the impact of large second level caches on the relative performance of the VM-based schemes. The performance of Snoop with the larger caches improves by 10%, 17%, 22%, 18%, 67% and 70% for LOCUS, WATER, MP3D, CHOLESKY, GAUSS and MUL respectively. The improvements with the VM-based schemes are not as high. The performance gains with VM-RC are 5%, 10%, 6%, 7%, 14%, and 58% respectively.

Figure 11 is used to analyze the behavioral pattern of the VM-based schemes. The first thing to notice is that larger caches make the relative performance of the no-caching approach much worse with respect to Snoop. The extreme is shown by MUL, where not-caching actually performed better then Snoop for the small cache, but performs significantly worse with the larger caches. The main reason for the difference in relative performance is that, the larger caches allow Snoop to reduce the number of cycles due to memory accesses significantly; the cache miss ratio with Snoop is reduced anywhere from 23% (for MP3D) up to 93% (for GAUSS) by using the larger cache.

Not caching also performs significantly worse than the VM-based schemes, but the differences vary among applications. For LOCUS, WATER, MP3D and CHOLESKY the relative

performance of the VM-based schemes stays fairly constant when compared to Snoop and therefore is much better than NO. Because, in relative terms, the cost of coherence induced cache misses is higher with the VM-based schemes for these applications, the relative performance of the VM-based schemes is generally slightly worse with larger caches. However, for these four applications, the VM-based schemes remain within 33% of the hardware snoopying approach.

For the other two applications, GAUSS and MUL, the relative performance of NO is much worse with larger caches, and the performance of the VM-based schemes seems to be proportional to the change in relative performance of NO. Although for both schemes the relative performance of the VM-based schemes with respect to Snoop deteriorates significantly, the difference is that for GAUSS this means that the VM-based algorithms actually perform much worse than Snoop (within 130% rather than -10% with the smaller cache), while for MUL the VM-based approaches are still better than Snoop, but with a much smaller margin (6% rather than 35%). It is obvious that the reason why GAUSS' performance deteriorates is the time idle due to write-buffer stalls.

When most of an application's working set fits in the cache, misses occur mainly due to coherence constraints. In this case, if the cost of a coherence miss is high, such as in the VM-based schemes, the relative performance is worse when compared to an optimized hardware approach. When, on the other hand, cache misses occur regularly due to capacity limitations, which are less expensive on architectures with low contention interconnection networks, the performance gap between the hardware and VM-based approaches closes significantly.

For very large second level caches, capacity misses should be reduced significantly, and coherency misses will dominate the cache performance. In this case the cost of the VM-based algorithms might not be offset by the benefit of multiple paths to the memory modules, particularly for applications such as GAUSS whose memory accesses get serialized at the memory modules and can therefore not take better advantage of the multiple paths to memory. Again, these applications would benefit highly from improved write-buffer techniques, such as coalescing.

5.5 Effect of Micro-kernel implementations

In micro-kernel implementations of operating systems the overhead introduced by the interface between the kernel and higher-level OS services, such as page fault management, is substantially higher than kernel traps.

Draves et. al presented an efficient way of implementing thread management for micro-kernel operating systems based on continuations [15]. This technique also provides more efficient means of implementing IPC in such operating systems. The runtime cost of components in the micro-kernel implementations were: 182 instructions to switch to a different thread in the kernel using continuations, and 341 using context switches. A null RPC consumed about 1500 cycles using continuations in the Mach MK40 micro-kernel on a DECstation 3100 (with a MIPS R2000 chip clocked at 16.67 MHz), compared to about 1800 cycles for the context switching implementation of the kernel.

To evaluate the impact of micro-kernel overheads on the VM-based cache coherence schemes we simulated an additional cost component per page-fault. On entry and exit of the fault handler we add the cost of the overhead of switching to the higher-level page-fault server. We measured the effect of executing this switch with low and high latency. We set the low latency value to 150 cycles, which is probably very close to an optimal

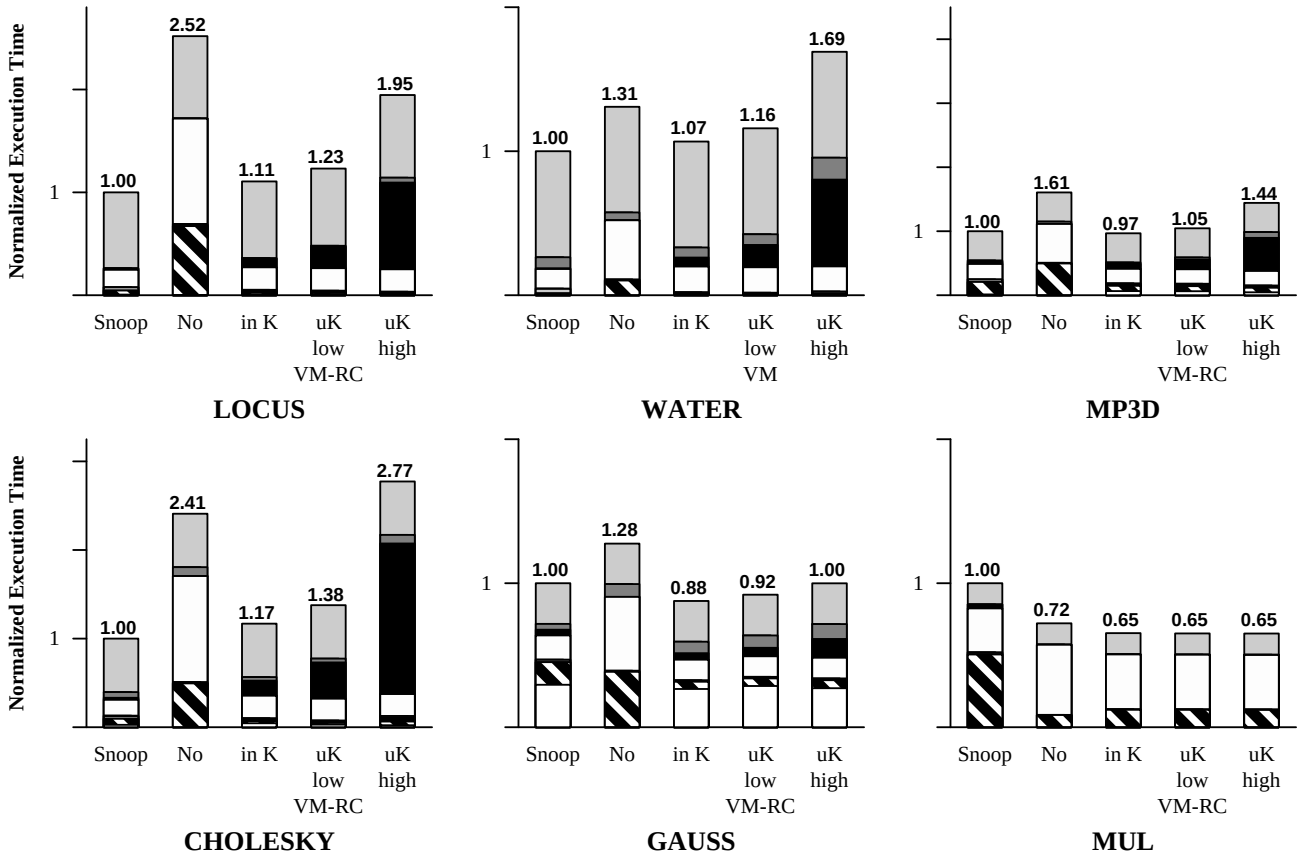

Figure 12: Effect of micro-kernel overheads.

implementation since it is even a little better than the 181 instruction count presented by Draves et. al. At the other extreme, we set the high overhead to 900 cycles, which is half of the full overhead of the null RPC measured for the context switching implementation of the Mach micro-kernel.

Figure 12 shows that very high latency IPC implementations of micro-kerneled operating systems would deteriorate the performance of the VM-based cache coherence schemes very much, but any other operating system services would be affected similarly. On the other hand, efficient implementations of micro-kernels would support VM-based cache coherence well.

5.6 Page Based Cache Invalidation

An expensive operation in the VM-based cache coherence algorithms is the invalidation of cache lines when the protection of a page changes from READ or WRITE to NIL. Some cache architectures, such as the MC88200 [6], allow all cached lines of a page to be invalidated with one instruction. For cache architectures that do not provide page selective cache line invalidation instructions, the approach of looping over or inlining the invalidations for all cache lines belonging to a page can be inefficient. For example, on architectures such as the Intel i860, the cache line invalidation and flush instructions are performed by a coprocessor, consuming several cycles for each operation. If one such operation has to be issued for every cache line of a page, the invalidation or flushing process can become too

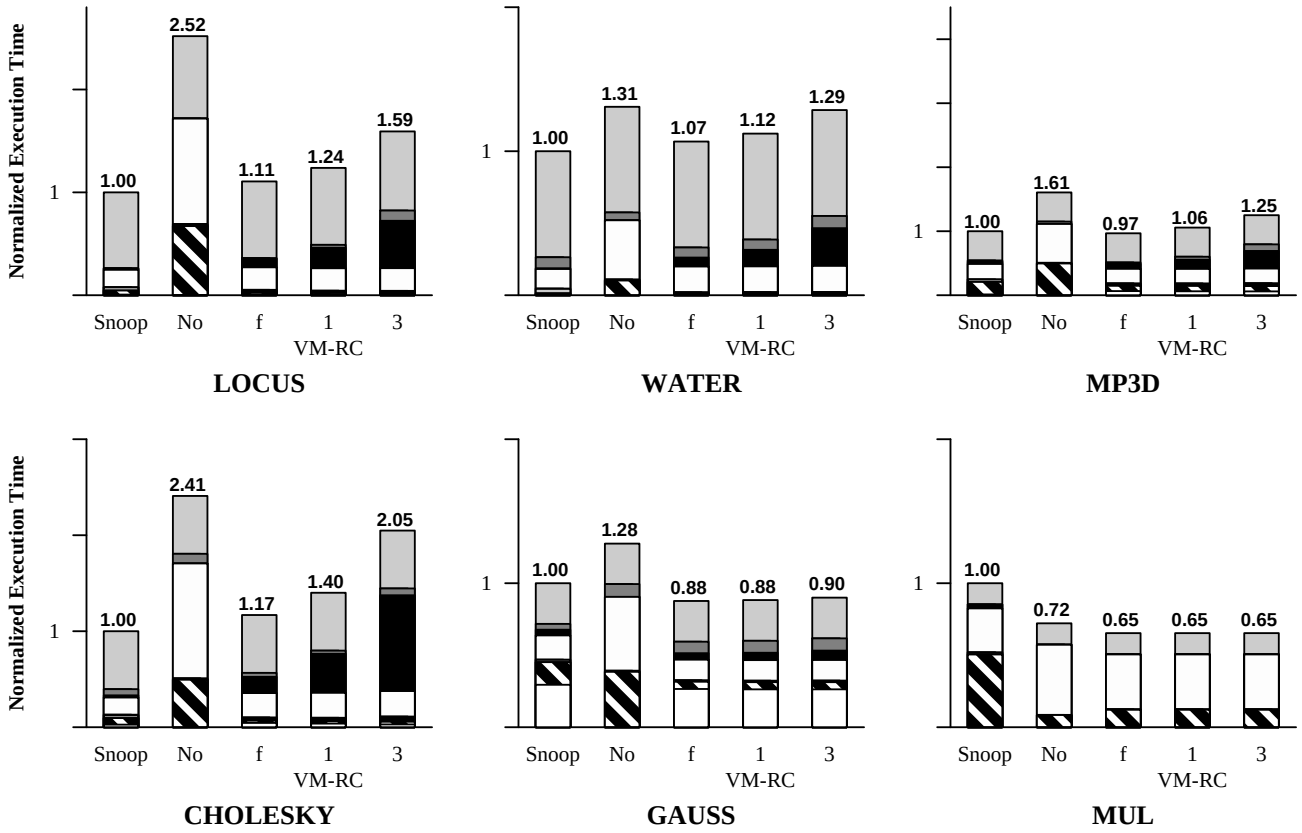

Figure 13: Effect of slow page based cache line invalidations.

expensive. To overcome this problem, we use a software mechanism for fast page-based cache invalidation.

The fast cache invalidation mechanism, we used in our simulations, is based on the notion of aliases for physical memory addresses. Aliases reference the same physical memory location, but have different physical cache tags. Therefore, a cache hit with one alias address will be a miss with another alias address, even though they refer to the same physical memory location. Physical address aliases were implemented with special TLB and cache hardware for the one-time-identifier scheme discussed by Smith in [44]. We reserve the unused high order bits in physical memory addresses to create physical address aliasing, without special hardware support. The idea of taking advantage of unused physical address bits for aliasing was first proposed by Greenstreet and Li to maintain cache consistency in the presence of dual-ported memories [22].

This alternative for invalidating all cache lines in a page, relies on the fact that caches are physically tagged. Although this is generally the case on current CPU architectures, here we also evaluate the impact on the VM-based cache coherence schemes, when the selective invalidation mechanism based on unused physical address bits cannot be used, as for example in virtually tagged caches. Figure 13 shows three bars for each VM-based scheme:

- *f:* The fast per page selective invalidation.
- *1:* When each cache line in a page needs to be invalidated separately and the cost of such an instruction is one processor cycle, and

- *3:* when the overhead of a one cache line invalidation is three processor cycles.

For almost all applications the three cycle overhead deteriorates the performance of the VM-based schemes significantly, up to 75% for CHOLESKY. This indicates that the performance of the VM-based schemes for an architecture with unsatisfactory support for cache line invalidations will not be very effective. On the other hand, for applications like LOCUS and CHOLESKY, which at times have a significant part of their virtual address space on the `WeakList`, it might be more effective to invalidate the whole cache, rather than probing every cache line belonging to pages on the list.

Although the one cycle overhead per cache line also reduces the effectiveness of the VM-based schemes, the maximum slow down is 20% for CHOLESKY and for all other applications it is less than 12%. The VM-based cache coherence schemes would still perform reasonably well on architectures with virtually tagged caches and minimal overhead to invalidate a cache line.

For the results of MP3D and CHOLESKY with VM-SC, longer cache invalidations have a curious effect on memory latencies. For both applications, the contention on the interconnection diminishes slightly, because the time it takes to invalidate all cache lines for a page is longer, and therefore the amount of busy waiting on synchronization variables and algorithm locks is reduced. In the case of MP3D, however, the reduction in contention is offset by the increase in algorithm overhead.

5.7 Scalability of VM-RC

Figure 14 shows the performance of the VM-based cache coherence protocols on architectures with four, eight and 16 processors. Each of the processors in the larger systems has the same characteristics as in our original simulations: 20 Kbyte on chip caches, 8 entry write-buffer, and low latency interrupts. For all cache coherence schemes, the processors are connected using a crossbar with a 29 cycle latency for cache refills in the absence of contention. Snoop is the only exception, it uses a bus with the same latencies. In the graph, the four processor execution time for the snoopy scheme represents 100%, and the execution time of all other schemes is normalized to it.

The results for the VM-based implementations of RC show that it scales to eight processors in a manner similar to Snoop. For some applications the relative performance of the VM-based schemes with respect to Snoop degrades slightly. For example, the eight processor execution of LOCUS under VM-RC is within 15%, rather than 11%, of Snoop. For WATER it is 17% rather than 7%, and for GAUSS 5% rather than -2%. MP3D and CHOLESKY actually scale better with VM-RC than Snoop to the eight processor architecture. For both applications, the increased number of paths to the memory modules alleviates contention on the interconnect.

The performance of the VM-based RC algorithms scales better than Snoop's when moving to 16 processors. The performance improvement of Snoop with 16 processors is minimal compared to the eight processor system. The maximum reduction in execution time is 23% for WATER. For all other applications it is less than 13%. On the other hand, the execution times of all applications, but GAUSS, improve by over 26% in VM-RC. The VM-based RC algorithms scale better than Snoop because of the much increased memory bandwidth. GAUSS' performance is limited by the write-buffer activity, which as discussed earlier, does not benefit as much from the higher memory bandwidth.

It is important to note, that for applications with very high cache miss ratios, such as MUL, increasing the bandwidth to the memory modules is very important. Figure 14 shows

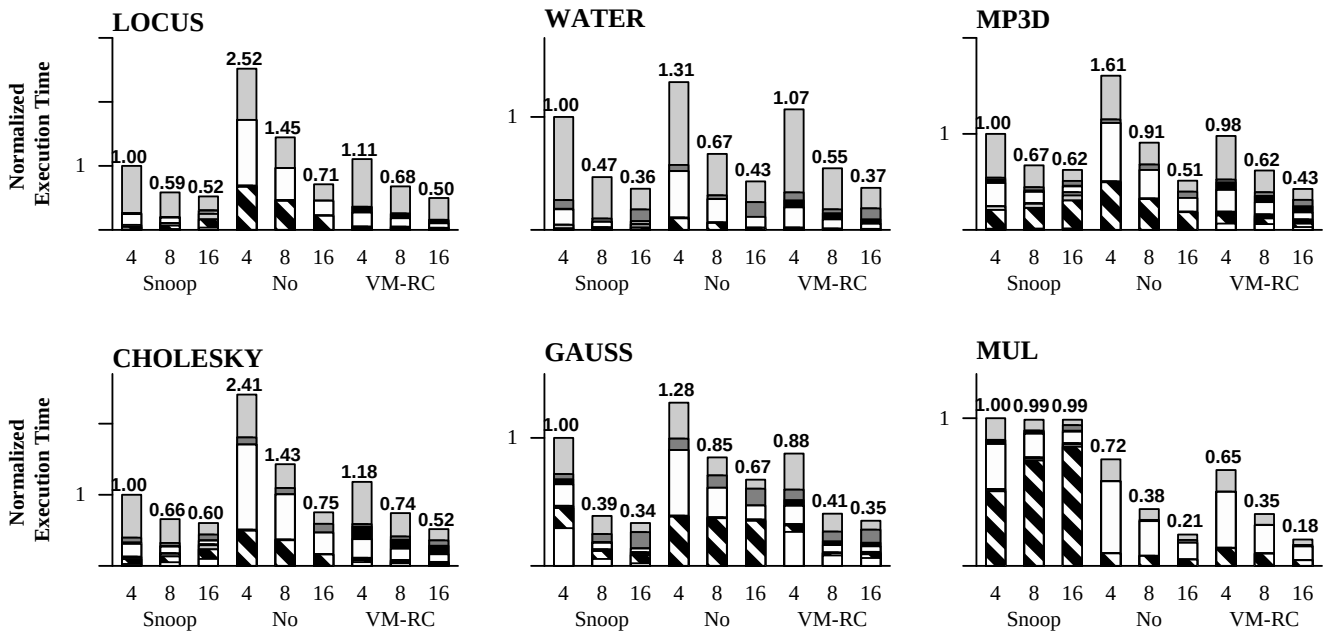

Figure 14: Performance with larger number of processors.

that MUL does not scale at all on the bus based architecture, because of the load on the interconnection.

6 Conclusions

VM-based cache coherence schemes provide an effective and economical alternative for building simple shared-memory multiprocessors. Our results show that the VM-based schemes can perform very well. VM-based RC with write-through performs best in most cases because the processors in the system are synchronized only when the application itself requires synchronization. The performance of VM-based RC stays within 30% of the snoopying hardware implementation for all applications evaluated in our study. Kontothanassis and Scott show that for larger scale architectures VM-based schemes stay within 50% of the best hardware implementations.

The VM-based cache coherence schemes can be implemented easily using write-through for cache lines containing shared data. However, the performance of the VM-based schemes is sensitive to the contention on the interconnection network. Using write-through for shared data can overload the interconnect if an application updates its shared data frequently. This problem is not solved by using write-back caches because the overhead of the algorithms and the burstiness of the memory update activity when flushes are requested, exceeds the benefit of lower network contention between flush requests. Coalescing write-buffers and code annotations can be used to reduce the impact of write through.

In general the simplicity of the architecture for the VM-based schemes has several advantages over hardware snoopy-cache schemes. It allows shared memory multiprocessors to be built with off-the-shelf uniprocessors and caches, keeping the development cycle to a minimum and allowing for more efficient use of chip real estate. However, the more significant advantage is that arbitrary interconnection networks such as crossbars can be used to connect processors to memory modules. For applications with high memory bandwidth

requirements our evaluation showed that performance can be increased by almost 40% with the VM-based schemes on crossbar-based architectures over a hardware cache coherence implementation on a bus-based machine. The scalability of the VM-based RC algorithms to a larger number of processors is also better than the one of snoopy cache protocols because of the much increased aggregate memory bandwidth supported by the VM-based schemes.

Another advantage of the VM-based schemes is that different consistency models can be used depending on the application's needs. The shared memory address space of parallel applications can be partitioned with respect to any criterion, and different coherence models can be used to manage each partition. This flexibility allows compilers and programmers to perform optimizations with data placement and coherence strategies easily. Furthermore, the parallel execution of multiple unrelated programs can take advantage of full uniprocessor performance without being encumbered by cache coherence hardware.

Several parameters affect the performance of the VM-based schemes. In this work we studied the effect of page size, cache size, latency of memory accesses, latency of interrupt handlers, and scalability on the relative performance of the VM-based schemes. With respect to page size, the results show that although large pages are prohibitively expensive for the VM-based implementation of sequential consistency, they can slightly improve the performance of VM-based RC.

Cache size affects the performance of the VM-based schemes as follows: when most of an application's working set fits in the cache, misses occur mainly due to coherence constraints. In this case, if the cost of a coherence miss is high, such as in the VM-based schemes, their relative performance is lower when compared to an optimized hardware approach. On the other hand, when cache misses also occur regularly due to capacity limitations, which are less expensive on architectures with low contention interconnection networks, the performance gap between the hardware and VM-based approaches closes significantly. For the applications we evaluated the performance of the VM-based approach with the large caches was still within 30% of the hardware approach.

Long memory latencies favor the relative performance of the VM-based approach because it can be implemented on non-bus based interconnections, therefore significantly reducing the contention on the network. The simulations of the longer memory latencies showed that the software approach never performed more than 10% slower than the hardware approach, and for three of the applications the VM-based approach actually performed 10–45% better than the hardware approach. When memory latencies increase mostly due to memory module set up time, and not because of network bandwidth, the relative performance of the VM-based schemes when compared to Snoop will depend on contention at the memory modules. Snooping mechanisms will benefit applications with high data locality because they support cache-cache communication, while the VM-based systems will benefit applications with less data locality by providing large aggregate memory bandwidth.

As long as the latency of an interrupt stays within reasonable ranges, its effect on the VM-based RC scheme is small, because the cost of executing an interrupt is amortized by the amount of processing that needs to be done during the page fault. Our results showed that efficient implementations of micro-kernel operating systems would support VM-based cache coherence well.

The algorithms presented in this paper are limited to medium scale multiprocessors because the `WeakList` is a centralized resource that can become a bottleneck as the number of processors in the system increases. As mentioned above, Kontothanassis and Scott have recently extended our work and have demonstrated the feasibility of VM-based schemes for larger scale machines.

Acknowledgements

This research was supported in part by NSF grant CCR-9020893, by DARPA and ONR under contracts N00014-91-J-4039, and by Intel Supercomputer Systems Division.

References

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 234–243, May 1991.

[2] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 298–307, May 1991.

[3] Sarita V. Adve and Mark D. Hill. Weak Ordering - A new definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, May 1990.

[4] James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 12th International Conference on Computer Architecture*, pages 355–362, June 1985.

[5] Anant Argarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherency. In *Proccedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, May 1988.

[6] Mich Aslup. Motorola's 88000 Family Architecture. *IEEE Micro*, pages 48–66, June 1990.

[7] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Proceedings of the IEEE COMPCON'93 Conference*, Feb 1993.

[8] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 782–789, August 1985.

[9] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 153–164, October 1991.

[10] David Chaiken and Anant Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 314–324, April 1994.

[11] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: A scalable cache coherent scheme. In *Proceedings of the 4th Internatinal Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[12] Alan L. Cox, Sandya Dwarkadas, Pete Keleher, Hongliu Lu, Rmakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[13] R. Cytron, S. Karlovsky, and K. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 229–238, August 1988.

[14] Ervan Darnell, John M. Mellor-Crummey, and Ken Kennedy. Automatic Software Cache Coherence through Vectorization. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 129–138, July 1992.

[15] Richard P. Draves, Briand N. Bershad, Richard F. Rashid, and Randall W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 122–136, October 1991.

[16] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[17] Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin McAuliffe, Larry Rudolph, Marc Snir, Patricia Teller, and James Wilson. Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 126–135, June 1985.

[18] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Paralle Processing*, volume I, pages 377–381, August 1991.

[19] Jean-Marc Frailong, Michel Cekleov, Pradeep Sindhu, Jean Gastinel, Mike Splain, Jeff Price, and Ashok Singhal. The next-generation sparc multiprocessing system architecture. In *Proceedings of the Spring IEEE COMPCON 1993*, pages 475–480, February 1993.

[20] Kouroush Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[21] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, June 1983.

[22] Mark Greenstreet and Kai Li. Cache Consistency with Dual-Port Memory: A Software Approach. Presented at the 2nd Workshop on Scalable Shared Memory Multiprocessors, May 1991.

[23] L. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, December 1989.

[24] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. In *Proceedings of the 5th Internatinal Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[25] Norman Jouppi. Cache write policies and performance. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 191–201, May 1993.

[26] Peter Keleher, Alan L. Cox, and Willie Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, May 1992.

[27] Leonidas Kontothanassis and Michael Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First HPCA*, pages 286–295, January 1995. *An extended version of the paper appears in this journal edition.*

[28] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, page ??, April 1994.

[29] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[30] R. L. Lee, P. C. Yew, and D. H. Lawrie. Multiprocessor Cache Design Considerations. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 253–262, June 1987.

[31] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The Stanford DASH Prototype: Logic Overhead and Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, May 1992.

[32] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, 7(4):321–359, November 1989.

[33] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[34] E. L. Lusk and R. A. Overbeek. Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors. Technical Report ANL-84-51, Rev. 1, Argonne National Laboratory, June 1987.

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

[36] John R. Nickolls. Overview of the KSR1 Computer System. Technical Report TR-9202001, Kendall Square Research, March 1991.

[37] Susan Owicki and Anant Agarwal. Evaluating the Performance of Software Cache Coherence. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, April 1989.

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

[39] Karin Petersen. *Operating System Support for Modern Memory Hierarchies.* PhD thesis, Princeton University, October 1993.

[40] 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, April 1994.

[41] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Consistency Schemes for MIMD Parallel Processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 340–347, June 1985.

[42] Silicon graphics inc. power series: 4d/480vgx-4d/310s. Technical report, 1991.

[43] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Computer Systems Laboratory, Stanford University, April 1991.

[44] Alain J. Smith. CPU Cache Consistency with Software Support and Using 'One Time Identifiers'. In *Proceedings of the Pacific Computer Communications '85*, pages 153–161, 1985.

[45] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings National Computer Conference*, pages 749–753, June 1976.

[46] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[47] Alexander V. Veidenbaum. A Compiler-Assisted Cache Coherence Solution for Multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 1029–1036, August 1986.

[48] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[49] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt. Mechanisms for cooperative shared memory. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 156–167, May 1993.