Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support

Karin Petersen Kai Li
Department of Computer Science
Princeton University

Abstract

This paper presents a software cache coherence scheme that uses virtual memory (VM) support to maintain cache coherency for shared memory multiprocessors and requires no special hardware to do so. Traditional VM translation hardware in each processor is used to detect memory access attempts that would violate cache coherence and system software is used to enforce coherence. The implementation of this class of coherence schemes is extremely economical: it requires neither special multiprocessor hardware nor compiler support, and easily incorporates different consistency models.

We evaluated two consistency models for the VM-based approach: sequential consistency and lazy release consistency. The VM-based schemes are compared with a bus based snoopy caching architecture, and our trace-driven simulation results show that the VM-based cache coherence schemes are practical for small-scale, shared memory multiprocessors.

Keywords: shared memory, multiprocessors, cache coherence, memory management unit, virtual memory, consistency models.

1 Introduction

Most previously proposed and implemented methods to maintain cache coherence in multiprocessor systems fall into two categories: hardware-based approaches, that require complete architectural support to implement the coherence protocols, and application-level software methods that require compiler support and often additional architectural support to improve performance. In this paper we present our evaluation of a new approach that requires neither special multiprocessor hardware, nor compiler support. It uses VM system level software to keep caches coherent and allows multiprocessors to be built from off-the-shelf uniprocessors, caches and arbitrary interconnections.

The most common hardware-based solutions use either snoopy caches [McC84, PP84, RS85, TS87, LT88] or directories [Tan76, ASHH88, LLJ+92]. Snoopy-cache protocols rely on a bus-based interconnection of the nodes in the system. Cache controllers listen to all transactions on this bus and take appropriate actions to keep caches coherent. Directory-based schemes use directory entries to keep track of cached copies for each shared memory block, and invalidate or update all copies upon updates to the memory block. 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 intrinsic problem: they require special purpose hardware that is expensive both in design time and chip real estate.

Application level software-based methods require compiler support to identify accesses to shared variables that might cause cache incoherencies. Veidenbaum presents a compile time algorithm that inserts cache invalidation instructions at parallel loop boundaries in [Vei86]. This algorithm was improved by Darnell, Mellor-Crummey, and Kennedy by moving aggregated cache invalidation instructions as close

as possible to parallel loop boundaries, but trying not to invalidate data unnecessarily [DMCK92]. In [LYL87] 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 [EGK+85] and the RP3 [BMW85] multiprocessor. Several forms of special hardware support have been proposed to improve the performance of these compiler assisted schemes such as using hardware to invalidate cache lines selectively [CV88, Che92], to support efficient invalidations [Smi85], and to reduce unnecessary invalidations [MB89].

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 and flush the cache are needed. Previous research [OA89, AAHV91] 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. On the other hand, compiler-based solutions are expensive because they have to be conservative when determining the interference of shared memory accesses, and require special hardware support for performance.

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

This paper reports the trace-driven simulation results of two memory consistency models: sequential consistency and lazy release consistency. We compare them with the Illinois invalidation-based snoopy-cache protocol [PP84], and to do so, we simulate a bus-based architecture and experiment with several programs written for snoopy-cache multiprocessors. Our simulation results show that the new software cache coherence methods are practical for small-scale multiprocessors, and that the performance of lazy release consistency for multi-threaded parallel programs can be close to the snoopy-cache coherence mechanism.

2 VM-based Cache Coherence

The idea of the VM-based cache coherence schemes stems from previous work on shared virtual memory [LH89]. This section describes the basic mechanism required for VM-based cache coherence schemes and the algorithms for implementing two memory consistency models.

Architecture and Mechanisms

VM-based cache coherence schemes can be implemented on very simple multiprocessor architectures. Off-the-shelf uniprocessors, uniprocessor physical caches, and standard memory modules connected through some interconnection network, as shown in Figure 1, will be sufficient. The only hardware support needed is an interprocessor interrupt mechanism over the network.

Memory management units (MMUs) are a standard component of most uniprocessor architectures. The implementation of the VM-based schemes requires an MMU to 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).

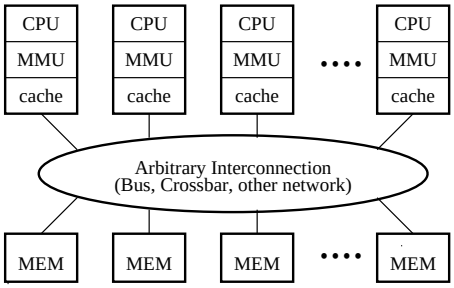

Figure 1: A simple multiprocessor using uniprocessor CPUs, caches, and memory modules.

- Physical caches which support selective *write-through* and selective invalidation of cached lines from software, as supported by architectures like the SPARC-1, the Intel i860, the MC68040 and the IBM RP3 [BF90, Int89, ELM90, BMW85]. In our algorithms we use write-through for shared data, but will also explain how to implement the VM-based schemes with *write-back* caches.
- Each processor has its own page table and TLB reloading can be managed from software.

The VM-based cache coherence software maintains coherency at the virtual memory address space level. When creating a shared address space, the system initializes a page table for each processor. In these tables, the entries (PTE's) 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, and will hereby reflect the coherency state of the page. VM software will maintain the page tables at run time according to a chosen cache consistency model.

Sequential Consistency

Sequential consistency (SC) [Lam79], 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. At a given point in time, sequential consistency allows only one writer per data item. In the absence of a writer, it allows multiple readers.

The VM-based implementation of SC is as follows:

Initial state: For each processor set the protection bits of all page table entries (PTEs) for shared pages to NIL.

Read fault on page i: Loop through every processor's PTE for page i. If a PTE has WRITE protection, interrupt its processor and change its PTE to READ. After checking all the PTE's, the faulting processor changes its PTE to READ and restarts.

Write fault on page i: Loop through every processor's PTE for page i. If a PTE has READ or WRITE protection, interrupt its processor, make it invalidate its cache lines for page i, then change its PTE to NIL. After checking all the PTE's, the faulting processor updates its PTE to WRITE, and restarts.

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.

Figure 3 shows an example of SC for a two processor system. Processors P_1 and P_2 have loaded data from page i, hence the protection bits for page i on both processors are set to READ, and the caches of both processors contain data from page i.

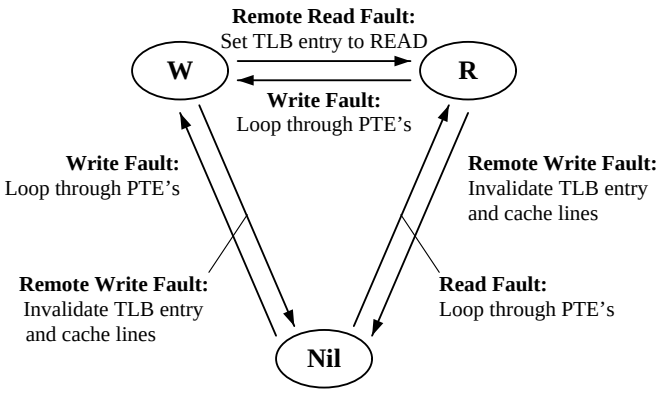

Figure 2: Transition diagram for sequential consistency.

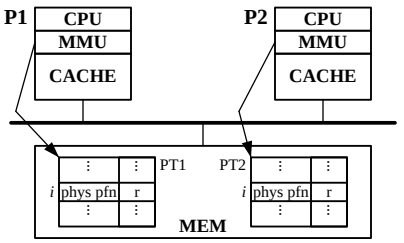

Figure 3: P1 and P2 read sharing page i

Suppose that processor P_2 performs a write to a memory location in page i. An access violation occurs and the page fault handler on P_2 fields the trap. The fault handler checks processor P_1's page table entry for page i, changes it from READ access to NIL, has P_1 invalidate the cache lines of page i, changes its own page table entry to WRITE, and then returns to the faulting instruction. The final state is shown in Figure 4.

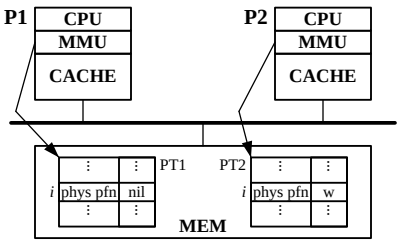

Figure 4: After P_2 writes into page i

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, we use one status word 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, i.e., 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 maintained consistent with the status words. For each shared address space the system will have a separate set of such shared page status words.

It is not hard to adapt the algorithm to use write-back caches instead of write-through caches. With

write-back caches, the memory may not have up-to-date data when another processor requests access to a page; our algorithm must therefore flush cache entries associated with a page when undergoing a transition from a WRITE state. These transitions now take longer, but overall traffic along the interconnection could be reduced.

Fast Cache Invalidation

The most expensive operation of the algorithm described above is the invalidation of cache lines when the protection of a page changes from READ or WRITE to NIL. To loop through all cache lines belonging to a page and invalidating them individually is inefficient. Some cache architectures, such as the MC88200 [Asl90], allow all cached lines of a page to be invalidated with one instruction. We propose a software mechanism for fast page based cache invalidation.

It is possible to postpone the invalidation of cache lines until the reverse page protection transition, that is, from NIL to READ or WRITE. We call such a method *lazy invalidation* because it only invalidates cache lines from a page before they need to become accessible. Lazy invalidation will not allow stale data to be accessed from the cache, because the PTEs protection set to no access will prevent the CPU from accessing the data without page-faulting. Although lazy invalidation avoids invalidations to pages that will never be accessed again, the cost for those that will be reaccessed would remain high.

The fast cache invalidation mechanism 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 [Smi85]. 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 [GL91].

In the fast invalidation mechanism, the unused high order bits serve as a page version number (equivalent to the one-time-identifier). Suppose that there are k unused bits and that $version_i$ is the version number for page i. Initially, $version_i$ is set to 0. When the protection of page i is changed from NIL to READ or WRITE, instead of invalidating the cache lines, the algorithm changes its PTE entry and sets $version_i = (version_i + 1) \bmod 2^k$. By incrementing the version number, the virtual page translation uses a different alias physical page number. The new physical address guarantees that the CPU sees no stale data in the cache for virtual page i, since no address translation will provide old physical addresses. When the value of $version_i$ is reset to 0, the system performs the expensive invalidations for all cache lines in the page.

The amortized cost of the fast invalidation method depends on how many unused physical address bits are available. With upcoming 64 bit machines, even if only $40 - 48$ of them are implemented for physical addresses, there should be enough unused bits in the addresses to make this scheme efficient.

Relaxed Consistency Models

Relaxed consistency models [DSB86, AH90, GLL+90] are based on the following observation: most shared memory multiprocessors guarantee that all memory accesses are globally performed at all times, i.e., a memory access completes only when all other processors see its results. However, most programmers only use this guarantee for synchronization variables (e.g. semaphores), and then use the ordering of these variables to ensure the correct use of other variables. Under relaxed memory consistency models, ordinary accesses can be issued and performed in any order, and synchronization points are viewed as fences that force memory accesses to be globally performed and therefore return the system to a coherent state. These

methods often reduce the access time of shared data structures significantly by hiding the latency of each memory access.

In addition to hiding latency, for the VM-based cache coherence schemes, relaxed consistency models also reduce the number of page faults caused by false-sharing. Figure 5 shows data access sequences on two processors. Assume that the variables a and b reside on the same page i. Right before time t_1, both processors have read access to the page. Under sequential consistency using the lazy invalidation method, when P_2 writes b, it generates a write fault and initiates the action to change P_1's PTE to NIL and P_2's PTE to WRITE. At time t_2, P_1 writes a. P_1 has a write fault and therefore P_2's PTE changes to NIL, P_1 invalidates it's cache lines for page i and changes it's own PTE to WRITE. At time t_3, P_2 loads a. P_2 has a read fault, P_1's PTE changes to READ, P_2 invalidates it's cache lines from the page, and P_2's PTE changes to READ. There are three page faults and two cache invalidations.

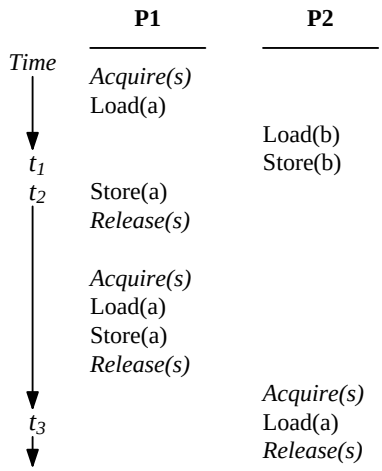

Figure 5: Example for relaxed consistency schemes.

Under a relaxed consistency model where loads and stores are ordered through `Acquire` and `Release` accesses, P_1 and P_2 can both have access to page i at time t_1 and t_2. There will be no page faults until P_2 performs the `Acquire` right before time t_3. In this example, a relaxed consistency model can eliminate two page faults and one cache invalidation due to false sharing.

Release Consistency

Release consistency (RC), as defined in [GLL+90], distinguishes between two types of synchronization accesses, `Acquire` and `Release`, and categorizes all other accesses as `ordinary`. RC is ensured by any system that adheres to the following rules when issuing memory accesses:

- Before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous `Acquire` accesses must be performed.
- Before a `Release` access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses much be performed.
- Synchronization accesses are processor consistent[1] with respect to one another.

[1] The conditions for processor consistency are: (1) load accesses stall for other load accesses to complete, but may bypass stores, and (2) stores can perform only if all previous loads and stores are performed.

A discussion on how to implement RC in hardware is described in [GGH91]. Here we will describe how to implement RC in software. Notice that 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 be delayed until the next `Acquire` of that lock. This is the main idea behind lazy release consistency (LRC) [KCZ91]. Intuitively, the advantage of LRC over RC is that memory updates need to be performed only when locks change owners.

[KCZ91] presents the following relation between shared memory accesses:

- If a_1 and a_2 are consecutive accesses in the same process, and a_1 occurs before a_2 in program order, then $a_1 \rightarrow a_2$.
- If a_1 is a `Release` access in process P_1, and a_2 is an `Acquire` access in process P_2 to the same synchronization variable, such that a_2 is the first `Acquire` on that variable to perform at P_1 after the `Release` of a_1, then $a_1 \rightarrow a_2$.
- if $a_1 \rightarrow a_2$ and $a_2 \rightarrow a_3$ then $a_1 \rightarrow a_3$.

Notice that the $\rightarrow$ ordering of accesses between distinct processes arises only from a `Release-Acquire` sequence between these processes. Based on the $\rightarrow$ relation, [KCZ91] defines a system to ensure LRC when it adheres to the following constraints:

- Before an ordinary (non-synchronization) LOAD or STORE access is allowed to perform with respect to any other processor, all previous (in $\rightarrow$ order) `Acquire` accesses must be performed with respect to that other processor.
- Before a `Release` access is allowed to perform with respect to any other processor, all previous (in $\rightarrow$ order) ordinary LOAD and STORE accesses much be performed with respect to that other processor.
- Synchronization accesses must be sequentially consistent with respect to one another.

Now we are ready to describe how VM-based LRC is implemented. We chose to make synchronization variables non cacheable, in order to avoid the page fault overhead on locks with high contention. Shared pages will therefore either contain ordinary data, and be cacheable, or synchronization variables, and then be non-cacheable. Most state of the art memory management units (MMUs) allow data to be cacheable on a per page basis; we use this property to mark pages that contain synchronization variables as non-cacheable, and all other pages as cacheable.

Synchronization variables are kept sequentially consistent, by ensuring that memory accesses block until completed at the memory modules. For this, 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 by assuming a write-through cache, when a `Release` performs, all stores issued before the `Release` are guaranteed to have been performed at the memory modules as well.

In LRC, page faults on ordinary data accesses will be used to keep track of how many processors have access (READ or WRITE) to a shared page, and when a shared page is being shared and updated, i.e., when it might become incoherent.

On `Acquires` a processor needs to invalidate the cache lines of shared pages to which it has access, that were also updated by any processor who previously held the lock. In order for a processor to know which pages are being simultaneously shared and updated we define 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` to which it has access, and performs the appropriate invalidations. A counter, $weak_i$, is used 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 LRC is as follows:

Initial State: For each processor set the protection bits of all page table entries for shared pages to NIL. In addition: `WeakList` = NULL, and $weak_i = 0$ for all i.

Read Fault on page i: Increment $weak_i$. If there is a processor writing page i then make sure i is in the `WeakList`. Set the PTE to READ and restart.

Write Fault on page i: Increment $weak_i$ if faulting from NIL access. If $weak_i > 1$ then make sure i is in the `WeakList`. Set the PTE to WRITE and restart.

On `Acquire`: For all pages $i \in$ `WeakList`, if the processor has READ or WRITE access to i, then invalidate the cache lines in page i, and set the access of i to NIL. Decrement $weak_i$ and if $weak_i = 0$, remove page i from the `WeakList`.

On `Release`: when the architecture includes a write-buffer, it has to drain before the `Release` returns.

Again, consider the execution sequences in Figure 5. At the first `Release` after t_2, page i's state is incoherent since both P_1 and P_2 have read access and updated variables from this page. Under strict RC, when P_1 releases s for the first time, it would have to make sure that all updates to page i were visible to P_2. With the VM-based approach this would require P_2 to invalidate its PTE corresponding to page i, as well as, all cache entries from page i.

In reality there is no need for P_1 to make sure that the store to a has been performed with respect to P_2 at the first release, and this is what LRC tries to exploit: when P_2 acquires s, it is guaranteed that all stores preceding the release of s by P_1 have been performed with respect to the memory modules. Recall that there are no entries in P_1's write-buffer that correspond to any of these stores. Therefore, it is enough for P_2 to invalidate all its PTEs and cache lines for pages in the `WeakList` before it completes with the `Acquire` of s at time t_3. In the example, P_2 will set its PTE for page i to NIL and invalidate the cache entries for b. Thereafter, an access to the variables updated by P_1, in the example the `Load(a)`, will miss in the cache and therefore the correct values will be loaded from memory.

In the example it becomes clear that stores issued by one processor perform with respect to another processor when lock ownership is transferred from one to the other.

3 Trace-Driven Simulation

We use trace-driven simulations to evaluate the proposed VM-based cache coherence methods. In order to compare the VM-based methods with a hardware-based cache coherence mechanism, we selected application programs implemented for snooping bus-based multiprocessors with per-processor caches and uniformly accessible shared memory. The applications were modified only to allocate synchronization variables on pages separate from other shared data. Other than that, the application's algorithms, memory allocation strategies, and data placement remained the same.

Applications

The application suite includes a Gaussian elimination program, matrix multiply, and three programs from the SPLASH parallel applications [WG89, SWG91]. All applications are coded in C, and use the *parmacs* macros from Argonne National Laboratory [LO87] to implement parallel operations such as: creation of cooperating processes, allocation of shared memory, and usage of lock and barrier synchronization variables.

LocusRoute, from SPLASH, is a parallel VLSI standard cell router. The basic strategy of the router is to minimize layout area, and to do so the program evaluates the cost of routing a wire through cells that have the smallest number of other wires running through them, and finally chooses the route of minimal cost. The work is parallelized by having all wires routed on a task queue, and processors cycle in a loop of removing a wire from the queue and placing it. The task queue is not a bottleneck because each routing task takes a while to complete.

WATER, also from SPLASH, is an N-body molecular dynamics application that evaluates forces and potentials in a liquid system of water molecules. These forces and potentials are calculated over a series of time steps separated by barrier synchronization. Work is parallelized by statically partitioning molecules over all processors. These molecules will be close in the data structure, but need not be close in physical space. Since molecules have more interaction with other molecules that are close in space, locality of reference could be compromised if the initial placement of the molecules in the data structures is not appropriate. For the input used in this experiment, molecules are ordered by their $3D$-coordinates, which are uniformly distributed. Locking is used to access the molecule data structures atomically.

MP3D, the third program from SPLASH, is a 3-dimensional particle simulator for rarefied fluid flow. It simulates the trajectories of fluid molecules as an object moves through the fluid and collides with the molecules. Over a series of time steps, MP3D evaluates the velocities and positions within a bounded physical space of all molecules. The work is parallelized by partitioning the set of molecules and statically scheduling each set on a processor. The locality of references to the data structures for the physical space cells is low, because of the interference caused by several processors accessing the same space cell during the same time step. The processes use barriers to synchronize between the different phases that make up one time step, and no locking is used on the data structures for the physical space because the probability of data races affecting the result is small.

GE is a Gaussian elimination with partial pivoting on an 80×80 double floating point matrix. Rows are assigned to processors statically. Synchronization occurs twice during each pivoting phase: once after the pivot has been determined, and once at the end.

MUL is a program for matrix multiplication of two 80×80 double floating point matrices. Each processor computes the results for a statically determined set of rows. The reason for using such a simple program is to have variations in the benchmark suite.

Application	**Shared Addr.** (K)	**Trace Size** (MB)	**Instructions** (x1000)	**Data Refs.** (x1000)	**Distribution of Data References** (%)			
					Total Reads	**Shared Refs.**	**Shared Reads**	**Sync. Refs.**
LocusRoute	228	101.95	43944	13370	75	14	11	.08
WATER	146	391.77	244593	56250	81	8	7	.06
MP3D	205	11.59	5143	1528	71	36	22	.03
MUL	150	11.8	19172	1573	97	65	65	0
GE	50	5.39	7912	825	64	63	42	.07

Table 1: Address Trace Characteristics.

Table 1 shows the characteristics of the address traces, which include all memory references starting

when the collaborating processes are created, and finishing when they all rejoin. The shared memory requirements for the SPLASH applications are based on the data from [SWG91].

Simulated Architecture

The multiprocessor architecture simulated in our experiments consists of 4 uniprocessor nodes connected to a memory unit through a split transaction bus (see Fig. 6). The main reason for choosing a bus, rather than another kind of network, is to compare the VM-based methods with a snoopy cache scheme.

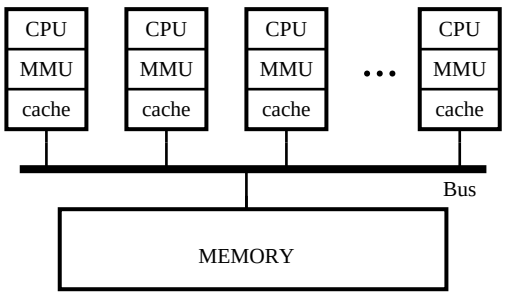

Figure 6: Simulated Architecture

Each node in the system resembles a MIPS R3000 processor, with a 64K direct mapped data cache. Writes are buffered in a 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. The memory unit consists of two interleaved memory modules and takes advantage of page mode.

The bus has separate address and data lines. The costs for the phases in a bus transaction are: 1 cycle for arbitration, 2 cycles for address latching, 2 cycles to send 4 bytes of data, 15 cycles latency at the memory modules for the first access to a page, and 2 cycles for subsequent accesses to the same page. The virtual memory page size is 1K bytes. The cache line size is 32 bytes. Therefore, the round-trip latency for a cache line fill in the absence of bus contention is 34 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.

We assume that every machine instruction takes one cycle. Although we do not simulate the instruction cache, instruction misses are artificially inserted with probability of .002 at each instruction. Cache lines are invalidated selectively using the fast invalidation method.

We have implemented a simulator to evaluate the different coherency schemes. The simulator works at the processor cycle level, and consists of a group of light-weight processes, each representing one of the processors being simulated. These processes cycle in a main loop reading from the trace files and executing the operations required to satisfy a trace-entry. Accesses to shared addresses are executed by calling a common set of routines provided by the modules that implement the different coherence schemes.

The synchronization primitives, `lock`, `unlock` and `barrier`, are simulated in the main loop. The simulator keeps track of the values of the synchronization variables and inserts accesses to the addresses of these variables in the trace to handle each synchronization event. For example, if processor P_1 tries to acquire a lock X, the simulator checks the value of lock X. If at this time X is held by another processor a sequence of instructions and memory accesses is inserted into P_1's trace to simulate P_1's retrying the acquisition of the lock.

The VM-based cache coherency algorithms are also simulated by dynamically adding instructions and memory accesses to the traces of each processor. We used the instruction sequence for the page access fault

handler in the Ultrix OS Kernel version 4.2 from the DEC-5000/200 workstations to estimate the cost for the instructions in the fault handlers of our algorithms.

Disk page faults are not simulated. This overhead is independent of the coherency schemes we are evaluating and should not affect our results.

4 Results

In this section, we present a comparative analysis of the performance achieved by the following cache consistency models:

- **Snoop** — The Illinois hardware invalidation-based coherence mechanism for snoopy caches.
- **NO** — No caching of shared data.
- **SC** — Sequential consistency model for the VM-based coherence scheme as described in Section 2.
- **LRC** — Lazy release consistency model for the VM-based coherence scheme as described in Section 2.

All models were simulated using the same architectural parameters.

Figure 7 shows the relative performance of the coherency mechanisms for each application. For the purpose of this paper we use normalized execution time 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 coherency mechanism.

To better understand the results, the graphs also show a breakdown of the execution time for all applications under each of the evaluated coherency methods. The breakdown is based on the average time one processor spends in each section. We identify four mayor regions in the breakdown: cycles a processor is idle due to bus 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.

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, it has poor performance for all applications.

Comparing NO with SC, we can see dramatic performance gains in all applications except for MP3D. There are two reasons for MP3D's behavior: it has low locality of reference, and a high degree of interference between processors accessing the same shared memory locations. The low locality is a result of statically assigning a set of molecules to each processor. Over time the molecules move into any space cells, and data interference occurs when molecules managed by different processors that move into the same space cells. In addition the granularity of our coherence unit, 1024 Bytes, adds to this factor by introducing high degrees of false sharing. Page faults occur at an average rate of 1 in every 15 accesses for the sequential consistency mechanism, of which 62% are false misses.

The performance gains of the VM-based SC approach over NO are considerable in all other cases, ranging from 20% to 55%. The application with the lowest performance gain is WATER, because it has the lowest degree of sharing. With an increasing degree of sharing, caching shared data structures becomes more critical. For example, with 14% of the memory references of LocusRoute being shared accesses, the

performance gain is 32%. For applications with an extremely large amount of shared data, such as GE and MUL, the performance gains are 45% and 55% respectively.

The performance gains by moving from NO to the VM-based schemes are very sensitive to the memory latency cycle time in the simulation. As the gap between microprocessor speed and DRAM speed continues to increase, we expect the performance gains by moving from NO to the VM-based schemes to increase as well.

LRC versus SC

LRC has the best overall performance. The performance gain over SC depends on the degree of contention for shared pages. For applications with very little contention for shared pages, such as MUL and Locus-Route, the difference is less than 10%, but for applications with more contention, the improvement can be substantial, e.g., 57% for MP3D.

For MP3D the overhead to keep coherency decreases from 30% of the total execution time in SC, to only 10% in LRC. This accounts for most of the improvement. In addition, by relaxing the consistency requirements, the cache miss ratio for MP3D decreases from 3.95 in SC to 2.47 in LRC. Both of these improvements also reduce the load in on the bus, allowing memory accesses to complete 50% faster in LRC.

WATER, on the other hand, actually performs a little better under SC. When moving from SC to LRC the number of page faults only decreases 2%, and the increased cost that LRC incurs when executing synchronization events outweighs the small page fault reduction.

LRC versus Snoop

The performance of LRC as compared to Snoop varies depending on the application. MUL performs best, within 2% of the snoopy cache approach. MP3D, however, requires 42% more time to execute under the LRC approach than under Snoop.

The execution time breakdown for each application shows two reasons for the difference in performance: (1) the overhead due to the VM-based implementation of LRC, and (2) the time a processors spend waiting for memory loads to complete.

The overhead of the VM-based implementation of LRC 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. We summarize this information in Table 2. For MP3D, who has the worst performance compared to the snoopy approach, the LRC overhead accounts for almost 15% of its execution time. For MUL, on the other hand, it is only .3%.

Application	Avg. WeakList Length	Coherency Synch. Events	Algorithm Overhead (% of total execution time)		
			Instructions	Mem. Acc.	Total
LocusRoute	18.6	170	5.0	3.3	8.3
WATER	4.2	2618	1.7	0.3	2.0
MP3D	30.6	35	10.0	4.9	14.9
MUL	2.0	2	0.2	0.1	0.3
GE	3.5	158	7.6	1.4	9.0

Table 2: LRC Overhead.

The other factor that contributes to the difference between LRC and Snoop, is the load on the memory bus. Although the number of bus transactions generated by accesses to private variables is roughly the same in LRC and Snoop, there is a substantial increase in bus activity due to shared accesses in LRC. This increase is only a little over a factor of one for MUL, about eight for MP3D, LocusRoute, and WATER, but as large as 28 for GE. For all applications but MUL, around 90% of all bus accesses to shared variables are due to cache write-throughs, only the remaining 10% is due to cache loads. Some additional bus traffic is also generated by memory accesses due to the LRC algorithm (shown in Table 2).

Scalability of LRC

Figure 8 shows the performance of VM-based LRC on an architecture identical to the above, with eight rather than four processors. Again, results are presented as normalized execution time relative to the snoopy cache implementation on four processors. The figure includes four different results for the performance of LRC on 8 processors. The pair of bars labeled LRC is generated from simulations with the page size parameter set to 1K bytes, which is the same as used in all previous results. The other pair, labeled LRC-8K, corresponds to simulations with 8K byte pages. For both LRC and LRC-8K we also show an estimate of the performance if a non-bus interconnection network is used to connect processors to memory modules.

The performance estimate for the non-bus interconnection was based on the assumption that bus load due to shared accesses can be evenly distributed on eight disjoint connections to each of the eight memory modules. We believe this assumption to be true, given that most of the shared memory traffic is due to write-buffer requests.

For applications, like WATER and MUL, with very little contention on their shared data, the VM-based LRC scales nicely to an eight processor system. WATER stays within 13% of the eight processor Snoop, and MUL, even closer, within 3%. The 1K byte page size, bus based execution of LocusRoute, with higher sharing activity (an average of 35 pages in the `WeakList`, that is, at least 30 higher than MUL and WATER), performs only within 36% of the eight processor execution of Snoop.

The 1K byte page bus based execution of MP3D and GE doesn't scale at all. It is obvious from Figure 8, that for both applications the bus is a major bottleneck. For MP3D the overhead of accessing memory amounts to over 60% of its execution time ($\frac{3}{4}$ of which is spend idle waiting for the bus or the write-buffer). With a non-bus interconnection MP3D would perform much better, and be within 46% of the Snoop scheme. GE's improvement with a non-bus interconnection is not as drastic because a considerable part (25%) of GE's execution time is spend for synchronization. Redesigning GE to spend less time synchronizing should be favorable to the LRC execution. For the MP3D version used in our experiments, [LLJ+91] shows that its speedups on the DASH architecture are poor as well, and a restructured version of MP3D (PSIM4) obtains better speedups on the same hardware. It should be safe to assume, that PSIM4 would also perform better under the VM-based implementation of LRC.

The results for LRC-8K show that for most applications, the advantage of having larger pages and therefore a shorter `WeakList`, outweighs the cost of possible additional page faults due to increased false sharing. Non surprisingly, MP3D shows the largest performance gain, 15% for the non-bus interconnection, since most of its pages become shared and updated during one of the phases of execution. In this case, by using 8K byte pages the number of page faults is also reduced. For GE, on the other hand, having larger pages actually deteriorates the performance of LRC. This is because larger pages don't significantly reduce the `WeakList` (from an average of 3 to and average of 2), but the amount of page faults increases.

All applications stay within 50% of the hardware snoopy cache scheme for a non-bus interconnection of the eight processor system. For LocusRoute, WATER and MUL, which don't have extreme contention for shared or synchronization data, this percentage is smaller than 20%.

5 Conclusions

VM-based software cache coherence schemes are effective and economical approaches to build simple shared-memory multiprocessors. Our results show that when contention for shared data structures is low, VM-based schemes can perform very well. Under low contention the best approach can perform as well as a snooping scheme. However, under extreme contention on shared memory accesses a 4 processor system stays within 42% of the snooping scheme.

The simplicity of the architecture for the VM-based schemes has a several advantages over hardware snoopy-cache and directory schemes. It allows shared memory multiprocessors to be build using off-the-shelf uniprocessors and caches, keeping the development cycle to a minimum. In addition, it allows for more efficient use of chip real estate and arbitrary interconnection networks such as crossbars.

Since VM-based schemes are implemented in software, different consistency models can be tested and realized without altering the hardware. Furthermore, the parallel execution of multiple unrelated programs can take advantage of full uniprocessor performance without being encumbered by cache coherence hardware.

Another advantage of the VM-based coherence methods is their flexibility. The shared memory address space of an application can be partitioned with respect to any criterion, and different coherence models can be used to manage each partition. The flexibility allows compilers and programmers to perform optimizations with data placement and coherence strategies easily.

Our results show that, the VM-based lazy release consistency model, LRC, in general performs better than sequential consistency, SC. In the one case LRC performs worse than SC, the difference is minimal. LRC is easy to implement on architectures with write-through caches.

For simplicity the current implementation of LRC maintains a central list of potentially incoherent pages. A single list has two disadvantages: it can grow large, and cause unnecessary invalidations. We are in the progress of evaluating further optimized approaches for lazy release consistency where the system maintains a list of potentially incoherent pages for each synchronization variable. We are also in the process of evaluating VM-based LRC implementations for systems with write-back caches. Applications, like MP3D, which generate high contention on the interconnection algorithms might profit from this approach.

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. Work on the trace collection was developed during Karin Petersen's summer internship at Matsushita Information Technology Laboratory. We thank Jim Plank for his participation in early discussions of the algorithms. We would also like to thank Mark Greenstreet and Anne Rogers for their careful comments on earlier drafts of this paper.

References

[AAHV91] 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.

[AH90] 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.

[ASHH88] 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.

[Asl90] Mich Aslup. Motorola's 88000 Family Architecture. *IEEE Micro*, pages 48–66, June 1990.

[BF90] Andreas V. Bechtolsheim and Edward H. Frank. Sun's SPARCstation 1: A Workstation for the 1990's. In *Proceedings of the Spring '90 IEEE COMPCON*, pages 184–188, February 1990.

[BMW85] 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.

[Che92] Hoichi Cheong. Life Span Strategy - A Compiler-Based Approach to Cache Coherence. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 139–148, July 1992.

[CV88] Hoichi Cheong and Alexander V. Veidenbaum. A Cache Coherency Scheme With Fast Selective Invalidation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 299–307, May 1988.

[DMCK92] 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.

[DSB86] 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.

[EGK+85] 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.

[ELM90] Robin Edenfield, Bill Ledbetter, and Ralph McGarity. The 68040 On-Chip Memory Subsystem. In *Proceedings of the IEEE Spring '90 COMPCON*, pages 264–269, February 1990.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the 3rd Internatinal Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[GL91] 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.

[GLL+90] 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.

[Int89] Intel. *i860 64-Bit Microprocessor - Advance Information.* Intel Corporation, 1989.

[KCZ91] Peter Keleher, Alan L. Cox, and Willie Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. Technical Report TR91-168, Computer Science Department, Rice University, November 1991.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *TOCS*, 7(4):321–359, Nov 1989.

[LLJ+91] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The Stanford DASH Prototype: Logic Overhead and Performance. Technical report, Computer Science Laboratory, Stanford University, December 1991.

[LLJ+92] 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*, May 1992.

[LO87] 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.

[LT88] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[LYL87] 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.

[MB89] Sang Lyul Min and Jean-Loup Baer. A Timestamp-based Cache Coherency Scheme. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume I, pages 23–32, August 1989.

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

[OA89] 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.

[PP84] 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.

[RS85] 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.

[Smi85] 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.

[SWG91] 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.

[Tan76] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings National Computer Conference*, pages 749–753, June 1976.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[Vei86] 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.

[WG89] 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.

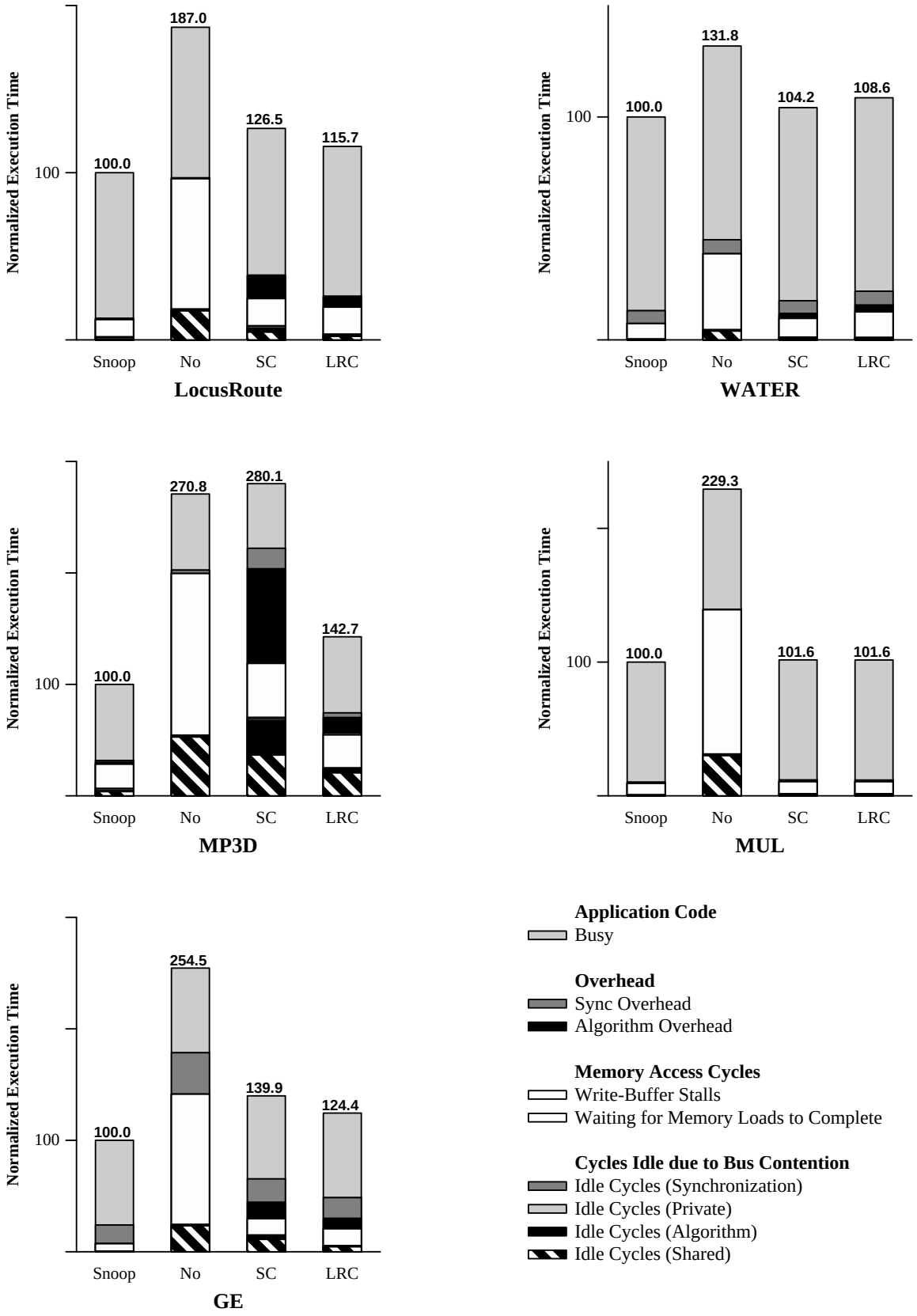

Figure 7: Performance of the coherency schemes for four processors.

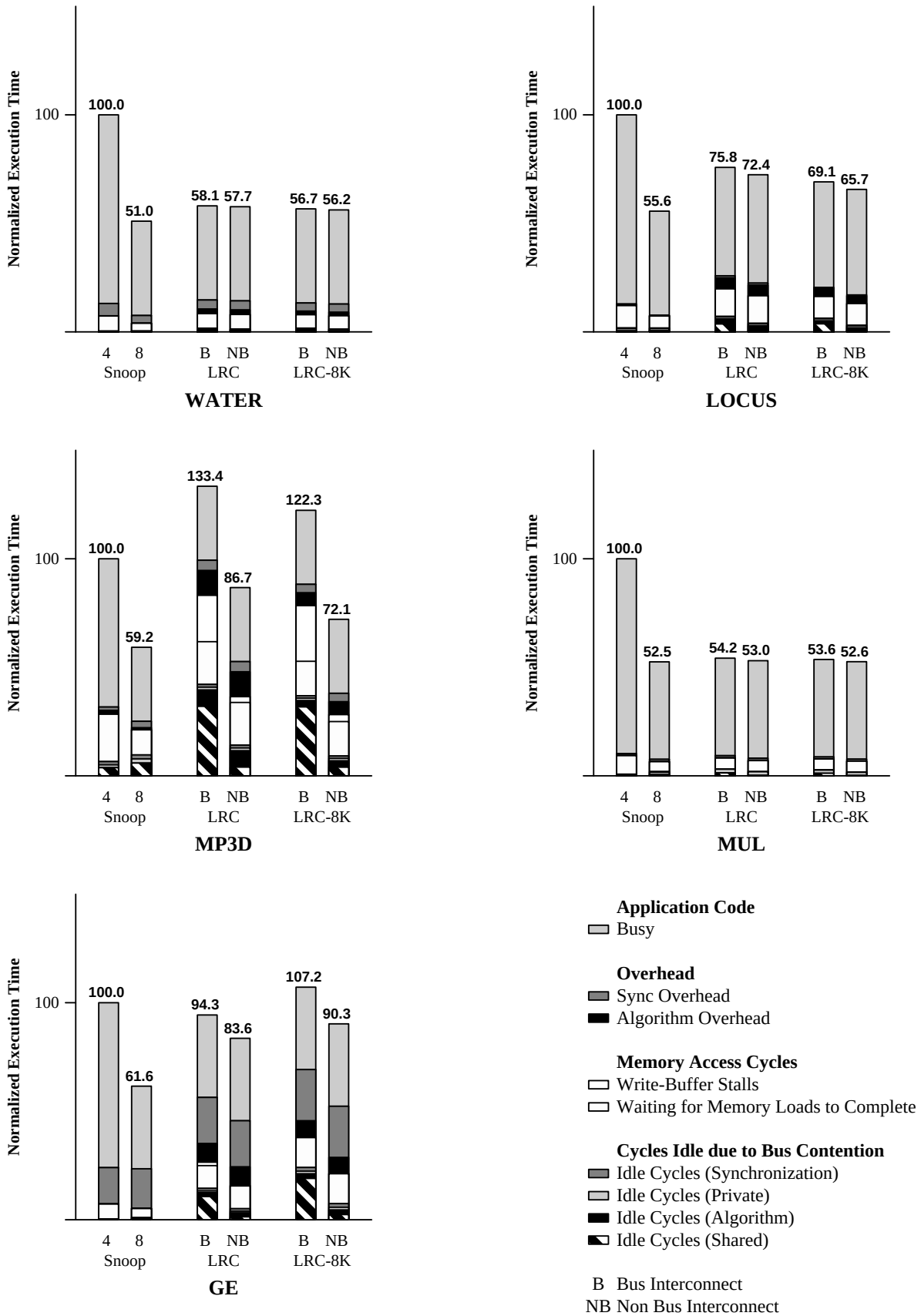

Figure 8: LRC versus Snoop performance for an eight processor system.