Comparison of Hardware and Software Cache Coherence Schemes[†,*]

Sarita V. Adve, Vikram S. Adve, Mark D. Hill, Mary K. Vernon

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

ABSTRACT

We use mean value analysis models to compare representative hardware and software cache coherence schemes for a large-scale shared-memory system. Our goal is to identify the workloads for which either of the schemes is significantly better. Our methodology improves upon previous analytical studies and complements previous simulation studies by developing a common high-level workload model that is used to derive separate sets of low-level workload parameters for the two schemes. This approach allows an equitable comparison of the two schemes for a specific workload.

Our results show that software schemes are comparable (in terms of processor efficiency) to hardware schemes for a wide class of programs. The only cases for which software schemes perform significantly worse than hardware schemes are when there is a greater than 15% reduction in hit rate due to inaccurate prediction of memory access conflicts, or when there are many writes in the program that are not executed at runtime. For relatively well-structured and deterministic programs, on the other hand, software schemes perform significantly better than hardware schemes.

Keywords: hardware cache coherence, software cache coherence, mean value analysis, workload model

1. Introduction

In shared-memory systems that allow shared data to be cached, some mechanism is required to keep the caches coherent. Hardware snooping protocols [ArB86] are impractical for large systems because they rely on a broadcast medium to maintain coherence. Hardware directory protocols [ASH88] can be used with a large number of processors, but they are complex to design and implement. An alternative to hardware cache coherence is the use of software techniques to keep caches coherent, as in Cedar [KDL86] and RP3 [BMW85]. Software cache coherence is attractive because the overhead of detecting stale data is transferred from runtime to compile time, and the design complexity is transferred from hardware to software. However, software schemes may perform poorly because compile-time analysis may need to be conservative, leading to unnecessary cache misses and main memory updates. In this paper, we use approximate Mean Value Analysis [VLZ88] to compare the performance of a representative software scheme with a directory-based hardware scheme on a large-scale shared-memory system.

In a previous study comparing the performance of hardware and software coherence, Cheong and Veidenbaum used a parallelizing compiler to implement three different software coherence schemes [Che90]. For

[†] This work is supported in part by the National Science Foundation (DCR-8451405, MIPS-8957278 and CCR-8902536), A.T.& T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation, and by an IBM Graduate Fellowship.

[*] Except for Appendix B, this paper appears in the Proceedings of the 18th Annual International Symposium on Computer Architecture, May 1991.

selected subroutines of seven programs, they show that the hit ratio of their most sophisticated software scheme (version control) is comparable to the best possible hit ratio achievable by any coherence scheme.

Min and Baer [MiB90a] simulated a timestamp-based software scheme and a hardware directory scheme using traces from three programs. They also report comparable hit ratios for the two schemes. However, they assume perfect compile-time analysis of memory dependencies, including correct prediction of all conditional branches, which is optimistic for the software scheme.

Owicki and Agarwal [OwA89] used an analytical model to compare a software scheme [CKM88] against the Dragon hardware snooping protocol [ArB86] for bus-based systems. They conclude that the software scheme generally shows lower processor efficiencies than the hardware scheme and is more sensitive to the amount of sharing in the workload. The main drawback of their method is that the principal parameters that determine the performance of the two schemes are specified independently of each other, and therefore *for a given workload* it is difficult to estimate how the schemes would compare. Furthermore, they assume the same miss ratio (0.4-2.4%) for private and shared data accesses in the hardware scheme, which is an optimistic assumption as shown in studies of sharing behavior of parallel programs [EgK89, WeG89].

Our analysis improves on the work by Owicki and Agarwal and complements the simulation studies by quantifying coherence protocol performance as a function of parameters that characterize parallel program behavior and compile-time analysis. Our model permits an equitable comparison of the software and hardware protocols for a chosen workload because we derive the principal parameters for each scheme from a common high-level workload model. Our workload model captures two important limitations of compile-time analysis that may reduce the performance of software schemes: 1) imperfect knowledge of runtime behavior, such as whether a write under control of a conditional branch will actually be executed, and 2) imperfect knowledge of whether two data references actually refer to the same memory location. Including compile-time and runtime parallel program characteristics in a unified model appears to be essential for comparing software and hardware cache coherence schemes.

From the high-level workload model, we derive two sets of low-level parameters that are used as inputs to queueing network models of the systems with hardware and software coherence. We compare a software coherence scheme similar to one proposed by Cytron et al. [CKM88] to a hardware directory-based Dir_iB protocol [ASH88] for large-scale systems. Our conclusions also hold for the version control and timestamp schemes, as discussed in Sections 5 and 6. The goals of our study are to characterize the workloads for which either the software or the hardware scheme is superior, and to provide intuition for why this is so.

The rest of the paper is organized as follows. In Section 2, we discuss the important issues that can result in performance differences between hardware and software schemes. In Section 3, we describe our common high-

level workload model. In Section 4, we first describe the system architecture and cache coherence schemes studied in this paper, and give a brief overview of the Mean Value Analysis models for the systems. We then describe how the low-level workload parameters are derived from the high-level workload model. Section 5 presents the results of our experiments. In Section 6, we discuss the overall results of our study, and comment on some related issues. Section 7 concludes the paper.

2. Performance Issues for Hardware and Software Coherence

In this section we outline the important issues that affect the performance of software and hardware cache coherence schemes. There are two main performance disadvantages of directory-based hardware schemes. First, substantial invalidation or update traffic may be generated on the interconnection network. Second, memory references to blocks that have been modified by a processor but not updated in main memory have to go through the directory to the cache that contains the block.

The performance of software schemes on the other hand is limited by the need to use compile-time information to predict run-time behavior. The limits of this information may force software schemes to be conservative when (1) predicting whether certain sequences of accesses occur at runtime, (2) using multi-word cache lines, and (3) caching synchronization variables.

To detect stale data accesses, the compiler has to identify sequences where one processor reads or writes a memory location, a different processor writes the location, and the first processor again reads the location. In this case, the compiler has to insert an invalidate before the last reference. To identify when such a sequence can occur, the compiler may need to predict some or all of the following: (a) whether two memory references are to the same location, (b) whether two memory references are executed on different processors, (c) whether a write under control of a conditional will actually be executed, and (d) when a write will be executed in relation to a sequence of reads. If any of these is not precisely known, the compiler has to conservatively introduce invalidation operations, perhaps causing unnecessary cache misses. Note that future advances in compiler technology could permit (a) and (b) above to be predicted accurately, while (c) and (d) involve runtime behavior that cannot be known at compile-time. In our analysis we explicitly model the problems of predicting whether and when a write is executed, and treat them separately from the first two sources of uncertainty in data dependence analysis listed above. In this context, we call a write that executes an *actual write*, whereas we say there is a *potential write* in the program when the compiler for the software coherence scheme has to insert an invalidate for reasons other than inaccurate prediction of memory access conflicts or processor allocation.

Another factor affecting the performance of hardware and software schemes is cache line size. For hardware schemes, it is an open question whether multiple-word cache lines provide higher performance than single-word lines for shared data. On the other hand, no software scheme proposed so far can use multiple-word lines to exploit spatial-locality for shared read-write data. Our workload model includes parameters to account for this factor.

Finally, all the software coherence schemes proposed so far require synchronization variables be uncacheable, whereas many hardware schemes allows such variables to be cached. In the future, the effects of this difference can be mitigated by software techniques [MeSar] that make locks appear more like ordinary shared data. For this reason, we do not model synchronization directly.

3. The High-Level Workload Model

Our high-level workload model partitions shared data objects into classes very similar to those defined by Weber and Gupta [WeG89]. We use five classes, namely, *passively-shared objects*, *mostly-read objects*, *frequently read-written objects*, *migratory objects*, and *synchronization objects*. Passively-shared objects include read-only data as well as the portions of shared read-write objects that are exclusively accessed by a single processor[1]. The latter type of data occurs, for instance, when different tasks of a Single-Program Multiple Data (SPMD) parallel program work on independent portions of a shared array.

Passively shared data generate no coherence traffic and hence do not cause performance differences between hardware and software coherence schemes.

We use the term *actively shared* to collectively denote all classes of shared data that are not passively shared. Table 3.1 summarizes the high-level workload parameters. (The column of values gives the ranges used in our experiments.) As discussed earlier, we do not model synchronization objects separately, but expect them to behave like ordinary shared data once contention-reducing techniques have been applied [MeSar]. The parameters for mostly-read, frequently read-written and migratory data are further discussed below. These parameters are designed to capture the sharing behavior of the particular data class, so as to reflect the performance considerations discussed in Section 2.

1. Note that this is a generalization of the read-only class defined by Weber and Gupta.

Table 3.1. Parameters of the high-level workload model.

Parameter	Value	Description
Parameters denoting the fractions of references to the various classes		
f_{data}	0.3	fraction of memory accesses that are data references
f_{pvt}	0.75	fraction of data references that are to private data
f_{PS}, f_{RW}, f_{MR}, f_{MIG}	0 - 1.0	fraction of shared references that are to passively shared, frequently read-written, mostly-read, and migratory data, respectively ($f_{PS}+f_{RW}+f_{MR}+f_{MIG} = 1$)
Parameters for mostly read data		
$f_{w \mid MR}$	≤ 0.1	fraction of accesses to mostly read data that are writes
l_{MR}	≥ 1	runtime average number of read accesses by a processor to a mostly-read data element between consecutive compiler-inserted invalidations executed on that element by the same processor.
n_{MR}	≥ 4	Mean number of processors that access a data element between consecutive (actual) writes to that element.
Parameters for frequently read-written data		
$f_{w \mid RW}$	0.1-0.5	fraction of accesses to frequently read-written data that are writes
l_{RW}	≥ 1	average number of read accesses by a processor to a frequently read-written data element between potential writes by other processors
n_{RW}	1-4	Mean number of processors that access a data element between consecutive (actual) writes to that element.
Parameters for migratory data		
l_{MIG}	≥ 2	average number of accesses to a migratory data element by a single processor before an access by another processor

3.1. Mostly-Read Data

Mostly-read objects are those that are written very infrequently, and may be read more than once by multiple processors before a write by some processor. An example is the cost array in a VLSI routing program which is read often by multiple processors, but written when an optimal route for a wire is decided. Even though actual writes to an object of this class are rare, there could be uncertainty in whether and when writes do occur, possibly causing a large number of unnecessary invalidations. We make the assumption that a processor always reads a mostly-read data element before writing it, so that a write always finds the data in the cache.

The parameters $f_{w \mid MR}$, l_{MR}, and n_{MR} describe accesses to mostly-read data, and are defined in Table 3.1. The feasible values of these three parameters are constrained in the following way. Define $ratio_{MR}$ to be the average number of compiler-inserted invalidates that a processor executes on a mostly-read data element in the interval between any two consecutive actual writes to the data element, averaged over the intervals when the processor does execute such invalidates. From the definition, $ratio_{MR} \geq 1$. Since a processor reads a data element l_{MR} times between compiler-inserted invalidates, $l_{MR} \times ratio_{MR} \times n_{MR}$ is approximately the total number of reads on a data element between two actual writes to the element.[2] But the latter is exactly the overall ratio of reads to writes at runtime, $(1-f_{w \mid MR})/f_{w \mid MR}$. Therefore,

2. The expression is approximate because the processors that perform the actual writes must be treated somewhat differently in the exact expression for the number of reads between a pair of actual writes.

$$ratio_{MR} = \frac{1 - f_{w|MR}}{f_{w|MR} \times n_{MR} \times l_{MR}} \geq 1. \quad (3.1)$$

This relationship is significant for two reasons. First, it relates the compile-time and runtime behavior of the program, and therefore the performance of the software and hardware coherence schemes for the given program. Second, it constrains the feasible parameter space to be explored in comparing the two schemes.

3.2. Frequently Read-Written Data

Frequently read-written objects are typically those that show high contention, such as a counter that keeps track of how many processors are waiting on a global task queue. Such data objects are written frequently, and also read by multiple processors between writes. Weber and Gupta show that this type of data can degrade system performance because they cause multiple invalidates relatively frequently. Writes to this type of data may also be executed conditionally, but a relatively high fraction of these writes would be executed compared to the mostly-read data. As for mostly-read data, we assume that a processor always reads a frequently read-written data element before writing it.

$f_{w|RW}$, l_{RW} and n_{RW} are defined in the same fashion as the corresponding parameters for mostly-read data (Table 3.1). By definition, the fraction of writes to this class, $f_{w|RW}$, is expected to be larger than $f_{w|MR}$. Also, n_{RW} is expected to be small. Similar to $ratio_{MR}$, we can define $ratio_{RW}$ and estimate it as

$$ratio_{RW} = \frac{1 - f_{w|RW}}{f_{w|RW} \times n_{RW} \times l_{RW}} \geq 1. \quad (3.2)$$

3.3. Migratory Data

Migratory data objects are accessed by only a single processor at any given time. Data protected by locks often exhibit this type of behavior, where the processor that is currently in the critical section associated with the lock may read or write the data multiple times before relinquishing the lock and permitting another processor to access the data. Migratory data resides in at most two caches at any time. Again, we assume that a processor always reads a migratory data element before writing it. For migratory data, l_{MIG} is the average number of accesses to a migratory data element by a single processor before an access by another processor.

4. Analysis of the Coherence Schemes

The high-level workload model described in the previous section is used to derive low-level parameters that are inputs to MVA models of the systems being compared. Before describing how the low-level parameters are

derived, we state our assumptions about the coherence protocols and the hardware organization, and give a brief overview of the Mean Value Analysis models.

4.1. System Assumptions and Mean Value Analysis

We assume a system consisting of a collection of processing nodes interconnected by separate request and reply networks, each with the geometry of the omega network, with 2×2 switches. We do not believe that the specific choice of network topology should significantly influence the qualitative conclusions of the study. Each node consists of a processor and associated cache, and a part of global shared memory. Messages are pipelined through the network stages. We assume that buffers are associated with the output links of a switch and have unlimited capacity, and that a buffer can simultaneously accept messages from both incoming links. The parameters describing the architecture are given in Table 4.1.

Table 4.1. Architectural and System Parameters.

Parameter	Value	Description
Architectural Parameters		
N	256	number of processors in the system
D_{sw}, D_{cache}, D_{mem}	1.0,1.0,4.0	no contention delay at switch (per packet), cache and memory respectively
L_i	2,8,10,2	number of packets in a message of type i, $i \in \{addr, data, addr+data, invalidate\}$
System Parameters		
ms_{ins}	0.005	fraction of references to instructions that miss
$ms_{pvt\&ps}$	0.01	fraction of references to private and passively-shared data that miss
loc_{hw}, loc_{sw}	1.0	reduction in miss rates to actively shared data due to spatial locality in the hardware and software scheme respectively
$cons$	0.1-1.0	reduction in hit rates to actively shared data due to conservative prediction of memory access conflicts and processor allocation.

For hardware coherence, we assume a simple directory-based Dir_iB protocol similar to the ones described by Agarwal et al. [ASH88]. A cache miss for a line in global *shared* state is satisfied by main memory, while a miss to a line in *modified* state is forwarded from main memory to the cache that owns the latest copy of the line, and this copy is returned directly to the requesting processor. On a write request to a line in shared state, invalidates are either sent from main memory to some average number of processors or are broadcast to all nodes in the system, consistent with a Dir_iB scheme. The requesting processor is not required to block for the invalidates to complete.[3]

As we will see, one situation where software coherence does better than Dir_iB is when a location is read and then written by a processor. This is because software can use one invalidate whereas Dir_iB may need to take

3. This implies that the system is not sequentially consistent.

Table 4.2. Low-level workload parameters.

Parameter	Description
Parameters for software cache coherence	
p_r, p_w	fraction of references that are read or write misses respectively
p_{post}, p_{inv}	fraction of references that are posts or invalidates respectively
Parameters for hardware cache coherence	
$p_{r\|sh}$, $p_{r\|mod}$	fraction of references that are read misses to lines in shared or modified state respectively
$p_{w\|sh}$, $p_{w\|mod}$	fraction of references that are write misses to lines in shared or modified state respectively
$p_{ind.inv.}$	probability that invalidations are sent individually (not broadcast)
n_{inv}	average number of processors to which invalidations are sent, when they are sent individually.

action on both the read and the write. This performance difference can be reduced if hardware supports a *Read-For-Ownership* (*RFO*) operation [KEW85]. *RFO* is a read operation that procures the requested line in modified state in the processor cache to avoid a directory access on a subsequent write. Since the use of *RFO* could significantly change the performance of Dir_iB relative to software coherence, we model Dir_iB without and with *RFO*.

For software coherence, we model a scheme similar to the one proposed by Cytron et al. [CKM88]. The compiler inserts an *invalidate* instruction before each potential access to stale data, causing the data to be retrieved from main memory. Also, if a write to a shared location is followed by a read by a different processor, the compiler inserts a *post* operation that explicitly writes the line back to main memory. We assume that the processor is blocked for one cycle for each *invalidate* and *post* instruction, i.e. we assume that the processor does not have to block for the post to complete. This is consistent with not requiring a processor to block for invalidates in the hardware scheme. *Read* and *write* misses are identical in behavior as far as the network and main memory are concerned.

We use similar approximate Mean Value Analysis models of the system for both coherence schemes. The shared hardware resources in the system, i.e., the memories and the interconnection network links, are represented as queueing centers in a closed queueing network. The task executing on each processor (representing a single customer) is assumed to be in ‘‘steady state,’’ executing locally for a geometrically distributed number of cycles between operations on the global memory. We assume that a global memory operation is equally likely to be directed to each of the nodes in the system, including the node where the request originates. The probabilities of various global memory operations per cycle comprise the low-level workload parameters and are defined in Table 4.2. These parameters are derived from the high-level workload model as explained in Section 4.2.

The MVA models used to calculate system performance are similar to models developed by others for the analysis of different types of processor-memory interconnects [VLZ88, WiE90]. The detailed equations of the model are given in Appendix B. These models can be solved very quickly and have been shown to have high

accuracy for studying similar design issues.

The performance metric we use is *processor efficiency*, defined as the average fraction of time each processor spends executing locally out of its cache. This measure includes the effects of hit rate and network interference in each of the schemes.

4.2. Deriving the Low-Level Workload Parameters

The low-level parameters for each coherence scheme are derived from the high-level workload model by calculating the probability that a reference of each class causes each type of global memory operation. The system parameters listed in Table 4.1 are used in this derivation.

For the shared-data classes, the global memory access probabilities are calculated assuming a one-word cache line size, and assuming accurate analysis of memory access conflicts. Then, to account for the reduction in miss rates due to spatial locality, these global memory operation probabilities are reduced by the factor loc_{hw} or loc_{sw}. Also, for the software scheme, the hit ratio of actively shared data is reduced by the factor *cons* to account for inaccurate prediction of memory access conflicts. The approach used in calculating the contributions of each shared class is described here, and the detailed equations for all the low-level parameters are given in Appendix A.

Mostly-Read Data. This type of data is read multiple times (l_{MR} times on the average) by a processor between compiler-inserted invalidates. The first read in each such sequence will be a miss for the software scheme, since it is preceded by an invalidate. Therefore, one in every l_{MR} reads to mostly-read data causes a miss in the software protocol. In the hardware protocol each write causes one read miss for each of n_{MR} processors, on the average. The probability of a read miss is therefore $n_{MR} \times f_{w\,|\,MR}$. Of these read misses, $1 / n_{MR}$ see the data in modified state (contributing to $p_{r\,|\,mod}$), while $(n_{MR}-1) / n_{MR}$ see the data in state shared (contributing to $p_{r\,|\,sh}$).

Writes to mostly-read data do not cause misses with the software protocol, because we assume that they follow a read access. However, each write causes a post operation. In the hardware protocol, all writes to mostly-read data contribute to $p_{w\,|\,sh}$. Furthermore, we assume that n_{MR} is large enough that broadcast is required for invalidations. This is consistent with Weber and Gupta's findings, which showed that writes to mostly-read data caused an average of 3 to 4 invalidates even for 16 processor systems [WeG89].

Frequently Read-Written Data. The contribution of this class to the probability of read and write misses is calculated in the same manner as for mostly-read data (when RFO is not included). Since this class has a relatively high fraction of actual writes, the assumption that each write finds the data in shared state will be somewhat pessimistic for the hardware scheme because two consecutive writes could be executed by the same processor, with no intervening reads by other processors. This assumption is also somewhat pessimistic for the software scheme,

since not all writes would cause a post operation.

Because fewer processors are expected to read between writes for this class (n_{RW} is low), we assume that all writes to data in shared state cause individual invalidates to be sent from main memory. Therefore, the contribution to $p_{ind.inv.}$ is the same as to $p_{w\,|\,sh}$. An average of $n_{RW}-1$ invalidates are required for each such write.

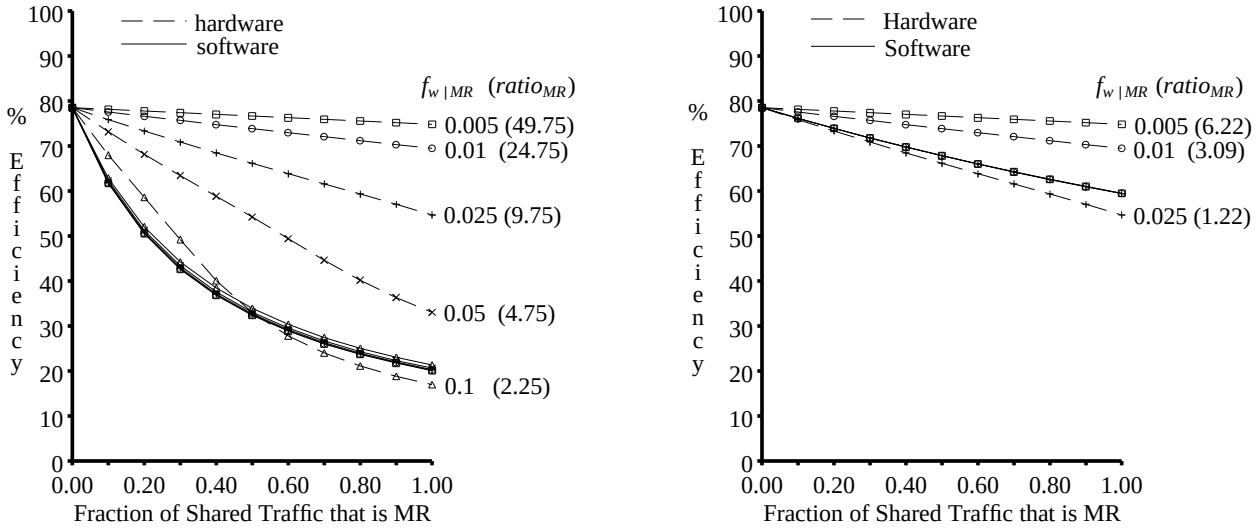
When RFO is included, every read sees the data in modified state, writes do not miss, and no invalidations are required.

Migratory Data. For migratory data, the first access in a sequence of l_{MIG} accesses is always a read by assumption. We assume that this type of data is written at least once for each sequence of accesses by a processor. Hence there is a read miss once per l_{MIG} accesses for both protocols. Therefore, for the hardware protocol, the first read by a processor in a sequence always finds the data in modified state. Writes in the software protocol do not miss since they always follow a read. In the hardware protocol without *RFO*, the first write of the sequence finds the data in shared state, causing a miss and causing an individual invalidate to be sent to exactly one processor. This miss and the invalidate are avoided, however, when *RFO* is included.

5. Results

We have used our models to perform experiments comparing the hardware and software coherence schemes. The constraints on the high-level workload model parameters discussed in section 3 (equations 3.1 and 3.2) allow us to explore the feasible workload parameter space completely. The ranges of workload parameter values that we consider reflect the characteristics of the shared data classes, and are given in Table 3.1. The system parameters (except *cons*) are held fixed throughout our experiments, and the values are given in Table 4.1. The values of f_{data} and f_{pvt} were chosen to reflect the findings of previous work characterizing parallel applications. [EgK88, OwA89]. Except for loc_{hw} and loc_{sw}, we believe that varying the other parameters will not affect the conclusions of our study. The value of 1 for loc_{hw} and loc_{sw} could be pessimistic for the respective schemes since they assume that spatial locality is not exploited. We will comment on these assumptions at the end of the section. Unless otherwise indicated, the experiments for hardware do not assume RFO.

In Sections 5.1 through 5.3, we study the effect of each class of actively-shared data in isolation, assuming $cons = 1$. In Section 5.4, we study the effect of smaller values of *cons*. Since the different data classes are independent of each other, their effects in isolation can be combined to draw conclusions about the overall performance of the software and hardware schemes. We discuss the overall performance results in Section 6.



(a) l_{MR}=1, n_{MR}=4 (b) l_{MR}=8, n_{MR}=4

Figure 5.1. Efficiency vs f_{MR} with varying $f_{w|MR}$.

5.1. The Mostly-Read Class

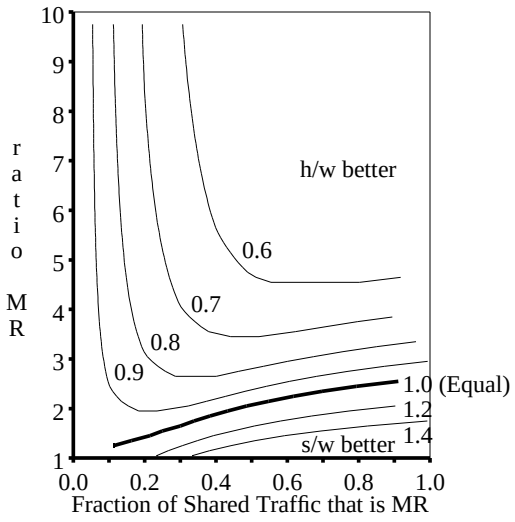
In figures 5.1(a) and (b), we plot the efficiency of the hardware and software coherence schemes as the fraction of shared data references that are to mostly-read data (f_{MR}) is varied from 0 to 1, while all other shared data is passively shared. The hardware scheme is sensitive to $f_{w|MR}$, the fraction of writes to mostly-read data at runtime, and n_{MR}, the mean number of processors that access a mostly-read data element between consecutive writes to the element. n_{MR} is held constant at 4 in both graphs, but the results are similar if $f_{w|MR}$ is held constant and n_{MR} is varied. The software scheme is sensitive to l_{MR}, the mean number of reads by a processor between compiler-inserted invalidates. Figure 5.1(a) shows the results for l_{MR}=1, the most pessimistic case for the software scheme. 5.1 (b) shows the results for l_{MR}=8, where the software scheme has become competitive with the hardware scheme.

In figure 5.1(a) we observe that as $f_{w|MR}$ increases, the efficiency of the hardware scheme decreases, while the effect on the software scheme is insignificant. Increasing $f_{w|MR}$ while holding n_{MR} and l_{MR} constant decreases $ratio_{MR}$, as shown in the figure. In effect, the number of potential writes in the program (and thus software performance) is held constant, while the fraction of these writes that are executed increases. An increased number of writes that are executed adversely affects the hardware performance in three ways: (1) each write that is executed is an additional miss, (2) the write results in broadcast invalidations causing higher network traffic, and (3) the first read by another processor after the write operation finds the line dirty and has to make an extra hop across the network to fetch the line.

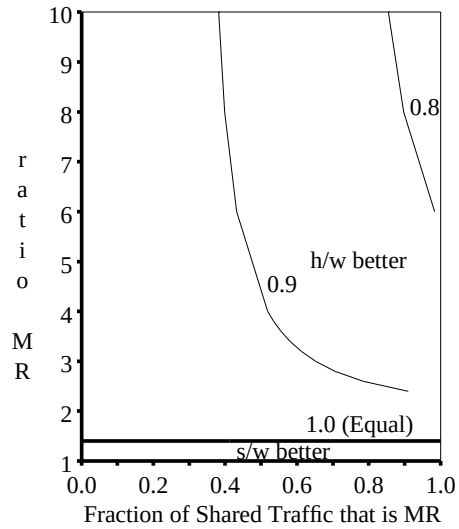
Figure 5.1(b) shows that the software scheme improves significantly for l_{MR}=8 as compared with l_{MR}=1, while the hardware scheme is independent of l_{MR}. Note here that the values $f_{w|MR}$ = 0.1 and $f_{w|MR}$ = 0.05 are not feasible for Figure 5.1(b) for n_{MR} = 4 and l_{MR} = 8, since they cause $ratio_{MR}$ to be less than 1. This restricts the region over which software would be superior to hardware.

We next identify the regions in the parameter space over which one of the schemes performs better. For l_{MR} = 1 and 8 in figures 5.2(a) and (b) respectively, we plot the contours of constant ratio of software to hardware efficiency over a range of values of $ratio_{MR}$, with the fraction of shared data references that are to mostly-read data varying from 0 to 1. In these experiments, $ratio_{MR}$ is varied by fixing n_{MR}=4 and varying $f_{w|MR}$. Similar results are obtained when $f_{w|MR}$ is fixed and n_{MR} is varied.

For low values of l_{MR}, we observe that the hardware scheme is significantly better (more than 20% better) than the software scheme if more than 20% of the shared data is mostly-read and $ratio_{MR}$ is greater than 3. In this case, the hardware scheme is superior to the software scheme for most of the feasible parameter space. Software coherence is more than 10% better than hardware only for very low $ratio_{MR}$. However, for $l_{MR}\geq 8$, the software scheme becomes competitive with hardware over most of the feasible parameter space.



(a) l_{MR}=1, n_{MR}=4.



(b) l_{MR}=8, n_{MR}=4.

Figure 5.2. Contours of constant $\frac{\text{Efficiency(software)}}{\text{Efficiency(hardware)}}$.

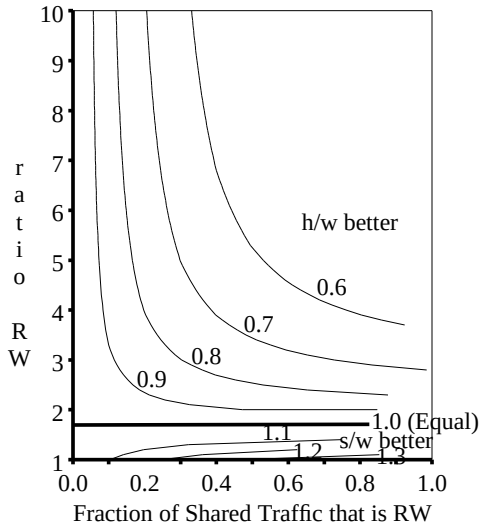


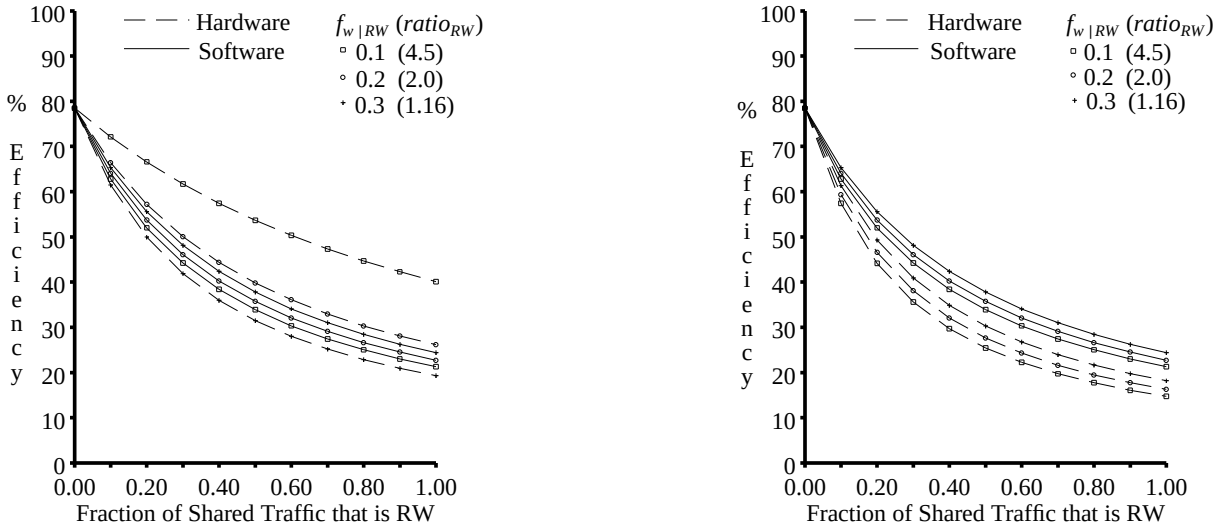
Figure 5.3. Contours of constant $\frac{\text{Efficiency(software)}}{\text{Efficiency(hardware)}}$, l_{RW} = 1, n_{RW} = 2..

5.2. The Frequently Read-Written Class

The parameters related to the frequently read-written class of data, l_{RW}, n_{RW} and $f_{w|RW}$, are similar to those for mostly-read data, but their values vary over different ranges, thus distinguishing the class.

The contour plots shown in Figure 5.3 give quantitative estimates of the relative performance of software and hardware coherence over the parameter space. As in figure 5.2, we use $ratio_{RW}$ to reflect the relationship between the behavior of the two schemes. Again, we vary $ratio_{RW}$ by holding n_{RW} = 2 and l_{RW} = 1 constant and varying $f_{w|RW}$. As for mostly-read data, the results are similar if n_{RW} is varied instead of $f_{w|RW}$. We observe that the hardware scheme is more than 20% better than the software scheme for $ratio_{RW} \geq 3$ and $f_{RW} > 0.3$. However, we expect that in many programs, less than 20% of shared data references would be to this class ($f_{RW} \leq 0.2$) since it leads to low processor efficiencies for any coherence scheme. Within this range of values, the software scheme is within 20% of hardware coherence in performance. For higher values of l_{RW}, the region for which software is comparable to hardware increases.

Since the RFO optimization may improve the performance of the hardware scheme for frequently read-written data, we examine how the relative performance of the two schemes changes with this optimization. The efficiencies for the cases without and with RFO are shown in Figures 5.4(a) and (b) respectively. Surprisingly, the RFO optimization degrades the performance of the hardware scheme, removing its advantage over the software scheme in regions where it dominates without RFO, for the entire parameter range that we explored. The reason for this counterintuitive result is as follows. Without RFO, only the reads that follow an *actual* write incur a miss,



(a) Without RFO. (b) With RFO.

Figure 5.4. The Effect of RFO with Frequently Read-Written Data.

requiring a global memory access, and only the first of these requires three traversals of the network. With RFO, every read incurs a miss for l_{RW} = 1 (here we assume that a data element is read by some other processor between successive writes by any processor), and requires three traversals of the network since the line is always held in modified state. When even a small fraction of the potential writes are not executed, the loss in efficiency due to the extra misses is not compensated for by the lack of misses when the writes occur, as shown in the plots for $ratio_{RW}$= 1.16 ($f_{w|RW}$ = 0.3).

Another point of interest is that, with RFO, the hardware and software schemes both have the same miss ratios for l_{RW}=1, but the software scheme has a lower cost per miss. In general, relative miss ratios do not completely reflect the difference between hardware and software schemes because of differences in network traffic and miss latencies.

5.3. The Migratory Class

The only parameter for migratory data is l_{MIG}, the average length of a sequence of accesses by a single processor. Figure 5.5 shows the contour plots for the relative efficiency of the software and hardware schemes with varying amounts of migratory data and l_{MIG}. The hardware schemes with RFO (solid lines) and without RFO (dashed lines) are shown. All other shared data is assumed to be passively shared. We observe that the hardware scheme consistently performs worse than the software scheme. This is essentially due to the deterministic behavior of this class of data. Without RFO, the difference is more than 20% for a large range of operation. The

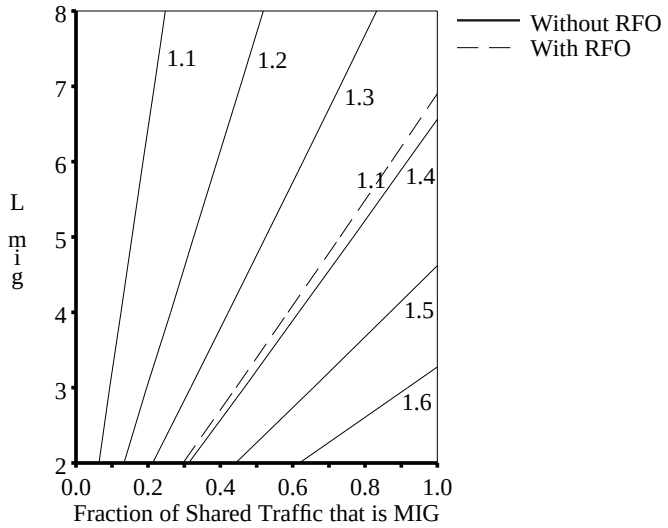


Figure 5.5. Contours of constant $\frac{\text{Efficiency(software)}}{\text{Efficiency(hardware)}}$.

RFO optimization brings hardware to within 20% of the software scheme over the entire parameter space, but does not make the hardware scheme outperform the software scheme. This is because, even though the use of RFO avoids the miss on the write for hardware, the read miss requires an extra hop to retrieve the data. Hence, the software scheme is always better than the hardware scheme for migratory data.

5.4. The Effect of Conservative Analysis of Memory Conflicts

The above experiments assume that conflicting memory accesses can be accurately identified at compile time. To analyze the effect of this assumption, we studied the effect of reducing hit rates to actively-shared data in the software scheme due to conservative analysis of conflicting accesses (*cons* < 1). Since the main difference between the hardware and software schemes occurs for mostly-read and migratory data accesses, we assume only these two classes of actively shared data in our experiments. Figure 5.6 plots the ratio of the efficiency of the software scheme to that of the hardware scheme with f_{MR} ranging from 0 to 1, and f_{MIG} = 1 - f_{MR}, with separate curves for different values of *cons*. The parameter settings used were those for which software had comparable performance to hardware coherence for *cons* = 1. We find that with up to about 10% reduction in hits due to conservative analysis (*cons*≥0.9), the software scheme stays within 10% of hardware. For more than 15% reduction in hit rate, the software scheme becomes more than 20% worse than the hardware scheme.

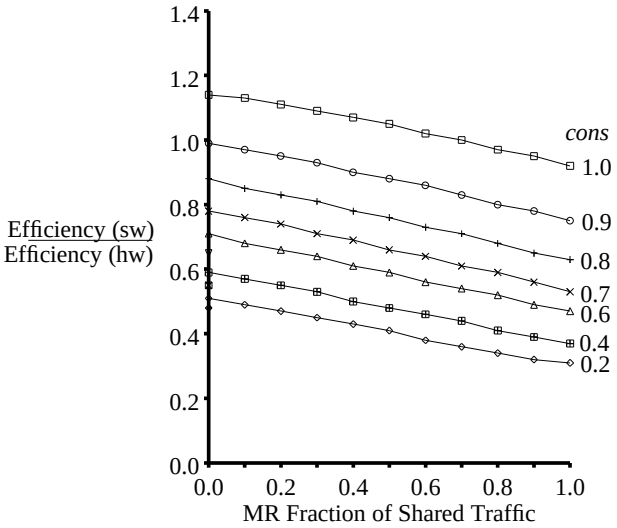


Figure 5.6. The effect of Conservative Analysis of Memory Conflicts, l_{MIG} = 4, l_{MR} = 8, $ratio_{MR}$ = 2, n_{MR} = 4.

6. Summary and Discussion of the Results

Our experiments show that if memory access conflicts can be detected accurately at compile time (*cons* ≥ 0.9), the software scheme is competitive with the hardware scheme for most cases. The most important case for which hardware coherence significantly outperforms software coherence is for the mostly-read class of data. With a high fraction of this class of data, if less than half of the potential writes detected at compile-time are executed, the hardware scheme can be more than 30% better than the software scheme. The hardware scheme is also significantly better with high fractions of frequently read-written data, when $ratio_{RW}$ is high. However, we do not expect parallel programs to contain such high proportions of this class of data. Otherwise, the software scheme performs within 10% of the hardware scheme for most cases. For migratory data, the software scheme consistently outperforms the hardware scheme by a significant amount. The RFO optimization for the hardware can substantially reduce this difference, but does not make the hardware scheme perform better than the software scheme.

The chief significance of these results is in showing the effect of various types of sharing behavior on relative hardware and software performance. For data that consists of conditional writes that are performed infrequently at runtime (high values of $ratio_{RW}$ and $ratio_{MR}$), the software scheme performs poorly compared to the hardware scheme. This suggests that if data with many conditional writes occurs frequently in parallel programs, some mechanism to handle these writes is essential for a software scheme to be a viable option. None of the software schemes proposed so far incorporate such a mechanism. Since the result of conditional branches cannot be predicted at compile time, some hardware support appears necessary so that the compiler can optimistically

predict branch outcomes, while the hardware takes responsibility for ensuring correctness when a prediction is wrong.

Although we have specifically modeled the scheme described by Cytron et al., we believe our results apply equally to the Fast Selective Invalidation scheme [ChV88] and to the timestamp based [MiB90b] and version control schemes [Che90]. The Fast Selective Invalidation scheme has been shown to be very similar to the Cytron et al. scheme in terms of compile time analysis and exploiting temporal locality. The timestamp-based and version control schemes have been shown to perform better than the scheme by Cytron et al., but our assumption of *cons* = 1 for the Cytron scheme makes it comparable to these more efficient schemes. Furthermore, neither of these schemes can effectively handle potential writes, and hence suffer as much from such conservative compile time predictions as the Cytron scheme.

Finally, all our results have assumed $loc_{hw} = loc_{sw} = 1$, i.e., neither scheme exploits spatial locality for actively-shared data. It is not known if software coherence schemes can effectively use multiple word blocks. It is also not known if multiple word blocks are desirable with hardware coherence schemes in large multiprocessors. If hardware schemes are shown to exploit significantly more spatial locality than the software schemes, our results no longer hold.

7. Conclusions

We have used analytical MVA models to compare the performance of software and hardware coherence schemes for a wide class of programs. Previous studies have yielded seemingly conflicting results about whether software schemes can perform comparably to hardware schemes. The conflict arises because the different studies make varying assumptions about the behavior of parallel programs.

We have characterized the workloads for which each of the two approaches is superior. There are two principal obstacles to such a study: (1) the sharing behavior of parallel programs is not well understood, and (2) for a specific workload, the relative performance of hardware and software schemes depends on the amount of runtime information that can be predicted at compile time. Our approach has been to use a high level workload model in which (1) shared data is classified into independent classes, each of which can be characterized by very few (1-3) parameters and studied in isolation, and (2) the relationship between the compile time and runtime characteristics is captured in a manner that can be related to the high level program, independent of the specific coherence scheme. The high level workload model is used to generate the workload parameters of the MVA model for each of the schemes, thereby allowing an equitable comparison of the schemes.

Quantitative data and intuitive explanations of the results were given in Section 5. The main conclusions of our study (assuming the software and hardware schemes exploit spatial locality equally) are as follows:

- Software schemes perform significantly less well (i.e., have at least 20% lower processor efficiency) than hardware schemes only if: (1) *cons* < 0.85, i.e., the hit ratio to actively-shared data is reduced by more than 15% because of conservative estimates of when two memory accesses conflict, or (2) less than half the potential writes are executed, on the average.
- Software schemes are comparable to hardware schemes (within 10% in terms of processor efficiency) if *cons* ≥ 0.9 and if more than half the potential writes in the program are executed.
- Software schemes are more efficient than hardware schemes, up to 20% better in some cases, if *cons* ≥ 0.95 and if most of the potential writes are executed.

Several important programs may fall under the category for which software coherence is significantly less efficient than hardware coherence. For example, detecting memory conflicts at compile-time for programs that make heavy use of pointers, such as operating systems and Lisp programs, could be difficult, i.e. *cons* would be low. On the other hand, for well structured deterministic programs, our results show that software schemes are comparable and in some cases better than hardware schemes. Many scientific programs fall under this class. Our study motivates the need for more work on characterizing parallel program workloads, and the relationship between compile time and runtime parameters of parallel programs. Once such a characterization has been made, our model and its results can be used more effectively.

Acknowledgements

We are grateful to Susan Owicki for many suggestions that have improved this work.

References

[ASH88] A. AGARWAL, R. SIMONI, M. HOROWITZ and J. HENNESSY, An Evaluation of Directory Schemes for Cache Coherence, *Proc. 15th Annual Intl. Symp. on Computer Architecture*, Honolulu, Hawaii, June 1988, 280-289.

[ArB86] J. ARCHIBALD and J. BAER, Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, *ACM Trans. on Computer Systems 4*, 4 (November 1986), 273-298.

[BMW85] W. C. BRANTLEY, K. P. MCAULIFFE and J. WEISS, RP3 Process-Memory Element, *Intl. Conf. on Parallel Processing*, August 1985, 772-781.

[ChV88] J. CHEONG and A. V. VEIDENBAUM, A Cache Coherence Scheme With Fast Selective Invalidation, *Proc. of the 15th Annual Intl. Symp. on Computer Architecture 16*, 2 (June 1988), 299-307.

[Che90] H. CHEONG, Compiler-Directed Cache Coherence Strategies for Large-Scale Shared-Memory Multiprocessor Systems, Ph.D. Thesis, Dept. of Electrical Engineering, University of Illinois, Urbana-Champaign, 1990.

[CKM88] R. CYTRON, S. KARLOVSKY and K. P. MCAULIFFE, Automatic Management of Programmable Caches, *Proc. 1988 Intl. Conf. on Parallel Processing*, University Park PA, August 1988, II-229-238.

[EgK88] S. J. EGGERS and R. H. KATZ, A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation, *Proc. 15th Annual Intl. Conf. on Computer Architecture*, Honolulu, HA, May 1988.

[EgK89] S. J. EGGERS and R. H. KATZ, The Effect of Sharing on the Cache and Bus Performance of Parallel Programs, *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, April 1989.

[KEW85] R. H. KATZ, S. J. EGGERS, D. A. WOOD, C. L. PERKINS and R. G. SHELDON, Implementing a Cache Consistency Protocol, *Proc. 12th Annual Intl. Symp. on Computer Architecture*, Boston, June 1985, 276-283.

[KDL86] D. J. KUCK, E. S. DAVIDSON, D. H. LAWRIE and A. H. SAMEH, Parallel Supercomputing Today and the Cedar Approach, *Science 231*(28 February 1986), .

[MeSar] J. M. MELLOR-CRUMMEY and M. L. SCOTT, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, to appear.

[MiB90a] S. L. MIN and J. BAER, A Performance Comparison of Directory-based and Timestamp-based Cache Coherence Schemes, *Proc. Intl. Conf. on Parallel Processing*, 1990, I305-I311.

[MiB90b] S. L. MIN and J. BAER, Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps, *Submitted for Publication*, 1990.

[OwA89] S. OWICKI and A. AGARWAL, Evaluating the Performance of Software Cache Coherency, *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, April 1989.

[VLZ88] M. K. VERNON, E. D. LAZOWSKA and J. ZAHORJAN, An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols, *Proc. 15th Annual Intl. Symp. on Computer Architecture*, June 1988.

[WeG89] W. WEBER and A. GUPTA, Analysis of Cache Invalidation Patterns in Multiprocessors, *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[WiE90] D. L. WILLICK and D. L. EAGER, An Analytic Model of Multistage Interconnection Networks, *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems 18*, 1 (May 1990), 192-202.

Appendix A

The method used to derive the low-level workload parameters from the high-level workload model is described here. For each low-level workload parameter, we describe the contribution to that parameter by each high-level actively-shared data class in a table. The entries in the table, when weighted by the probabilities of accessing shared data of that particular class, give the total contribution to that parameter by that class.

All private and passively-shared misses are included in p_r for the software scheme and in $p_{r\,|\,sh}$ for the hardware scheme. Let $T(param,c)$ denote the table entry in row *param* and column *c*. Then the equations used to derive the parameters for the software scheme are:

$$p_r = (1-f_{data})\, ms_{ins} + f_{data}\,(f_{pvt}+f_{PS})\, ms_{pvt\&ps} + 1 - cons \times \left(1 - \frac{f_{data}(1-f_{pvt})}{loc_{sw}} \times \sum_{c\in\{RW,MR,MIG\}} f_c\, T(p_r,c) \right) , \quad (1)$$

and for $y\in\{w,inv,post\}$,

$$p_y = 1 - cons \times \left(1 - \frac{f_{data}(1-f_{pvt})}{loc_{sw}} \times \sum_{c\in\{RW,MR,MIG\}} f_c\, T(p_x,c) \right) . \quad (2)$$

Similarly, for the hardware scheme:

$$p_{r\,|\,sh} = (1-f_{data})\, ms_{ins} + f_{data}\,(f_{pvt}+f_{PS})\, ms_{pvt\&ps} + \frac{f_{data}(1-f_{pvt})}{loc_{hw}} \times \sum_{c\in\{RW,MR,MIG\}} f_c\, T(p_{r\,|\,sh},c) , \quad (3)$$

and for $y\in\{w\,|\,sh, r\,|\,mod, w\,|\,mod\}$,

$$p_y = \frac{f_{data}(1-f_{pvt})}{loc_{hw}} \times \sum_{c\in\{RW,MR,MIG\}} f_c\, T(p_x,c) . \quad (4)$$

The table entries for the software scheme are given in Table A1, while those for the hardware schemes without and with RFO are given in Tables A2 and A3 respectively.

Table A1. Contribution of actively-shared data classes to software model.

Parameters of Software Model			
Param	Contribution of data class		
	MR	*RW*	*MIG*
p_r	$\frac{1-f_{w\mid MR}}{l_{MR}}$	$\frac{1-f_{w\mid RW}}{l_{RW}}$	$\frac{1}{l_{MIG}}$
p_w	0	0	0
p_{inv}	$\frac{1-f_{w\mid MR}}{l_{MR}}$	$\frac{1-f_{w\mid RW}}{l_{RW}}$	$\frac{1}{l_{MIG}}$
p_{post}	$f_{w\mid MR}$	$f_{w\mid RW}$	$\frac{1}{l_{MIG}}$

Table A2. Contribution of actively-shared data classes to hardware model without RFO.

Parameters of Hardware Model (no RFO)			
Param	Contribution of data class		
	MR	*RW*	*MIG*
$p_{r\mid sh}$	$(n_{MR}-1)f_{w\mid MR}$	$(n_{RW}-1)f_{w\mid RW}$	0
$p_{w\mid sh}$	$f_{w\mid MR}$	$f_{w\mid RW}$	$\frac{1}{l_{MIG}}$
$p_{r\mid mod}$	$f_{w\mid MR}$	$f_{w\mid RW}$	$\frac{1}{l_{MIG}}$
$p_{w\mid mod}$	0	0	0
$p_{ind.inv.}$	0	$\frac{f_{w\mid RW}}{p_{w\mid sh}}$	$\frac{1}{l_{MIG}p_{w\mid sh}}$
n_{inv}	0	$\frac{f_{w\mid RW}(n_{RW}-1)}{p_{ind.inv.}}$	$\frac{1}{l_{MIG}\,p_{ind.inv.}}$

Table A3. Contribution of actively-shared data classes to hardware model with RFO.

Parameters of Hardware Model (with RFO)			
Param	Contribution of data class		
	MR	*RW*	*MIG*
$p_{r\mid sh}$	$(n_{MR}-1)f_{w\mid MR}$	0	0
$p_{w\mid sh}$	$f_{w\mid MR}$	0	0
$p_{r\mid mod}$	$f_{w\mid MR}$	$\frac{1-f_{w\mid RW}}{l_{RW}}$	$\frac{1}{l_{MIG}}$
$p_{w\mid mod}$	0	0	0
$p_{ind.inv.}$	0	0	0
n_{inv}	0	0	0

Appendix B

The equations of the Mean Value Analysis model for the system with software coherence are given in detail here. The model for hardware coherence is similar.

We assume the processor cycle to be the unit of time for the model, and assume that a processor executes one instruction every cycle. Therefore, it makes $1+f_{data}/1-f_{data} = 1/1-f_{data}$ references every cycle. Defining $1/\tau$ as the probability that a processor requires a global memory operation or a coherence management operation (such as an invalidate or a post) in any given cycle, we have

$$\frac{1}{\tau} = \frac{p_r+p_w+p_{inv}+p_{post}}{1-f_{data}}. \tag{5}$$

We define $p_r' \equiv p_r/(p_r+p_w+p_{inv}+p_{post})$, and p_w', p_{inv}' and p_{post}' similarly. Then p_x' is the probability that a miss or coherence management instruction is of type $x \in \{read, write, inv, post\}$.

We can now calculate the *average response time* of a processor as:

$$R = \tau + p_r' r_r + p_w' r_w + p_{inv}' r_{inv} + p_{post}' r_{post} \ , \tag{6}$$

where r_r, r_w, r_{inv} and r_{post} are respectively the average response times for the respective operations, i.e. the average time the processor has to block for the operation to complete. Read and write requests are identical in behavior as far as the network and main memory is concerned. Therefore, $r_r = r_w$. We assume $r_{inv} = r_{post} = D_{cache}$.

The average response time for a read operation, r_r, is the sum of the network residence time and the main memory residence time for a read request. Since we assume pipelined routing, the network residence time is calculated by adding the weighted average residence times of the header packet of messages for all switch output ports in the system, and then adding one cycle for each packet of a message other than the header. To calculate the residence time at a switch output port, we identify two classes of customers at every port as explained below.

Since we assume a one-sided network, i.e., the global memory is distributed among the processing nodes, a switch output port on the path from a node to itself sees less traffic due to that node than due to the other nodes that can use the port. We call such a port an *own port* for that node. We call the other switch output ports that can be visited by that node *foreign ports* for that node. For every node, there is exactly one own port at any column of the network, and each output port is the own port for exactly one node. Also every node has the same number of foreign ports in a column and every output port in a column is a foreign port for the same number of nodes. A request from a node to its own port on a column is called an *own request* at that port, while a request to a foreign port on the column is called a *foreign request*. At each output port, we treat the own and foreign requests as two different classes of customers.

Before proceeding to calculate r_r, we introduce some notation. We denote the request or the forward network by *fd*, and the reply or the reverse network by *rev*.

$A^{y}_{j,i\,|x}$ ≡ queue length (not including the customer in service) of requests of type j seen by a request of class $x \in \{own, foreign\}$ arriving to a column i switch output port in the $y \in \{fd,rev\}$ network,

$P^{y}_{busy,j,i\,|x}$ ≡ probability that a request of class $x \in \{own, foreign\}$ arriving to a column i switch output port in the $y \in \{fd,rev\}$ network finds the switch busy serving a customer of type j,

$n^{y}_{sw,i\,|f}$ ≡ for any processing node, the number of *foreign* column i ports in the $y \in \{fd,rev\}$ network,

$n^{y}_{proc,i\,|f}$ ≡ the number of processing nodes that can make a *foreign* request to a column i switch output port in the $y \in \{fd,rev\}$ network,

$t^{y}_{i\,|x}$ ≡ mean residence time of the header packet of a message on an $x \in \{own, foreign\}$ column i port in the $y \in \{fd,rev\}$ network,

t_{mem} ≡ the mean residence time for a request at main memory,

$v^{fd}_{addr,i\,|x}$ ≡ probability that a read request visits an $x \in \{own, foreign\}$ column i port in the forward network as an address message; similarly, $v^{rev}_{data,i\,|x}$ for data messages on the reverse network.

Using the above notation, r_r can be expressed as follows:

$$r_r = \sum_{1}^{N_{col}} \left\{ \left(v^{fd}_{addr,i\,|o} t^{fd}_{i\,|o} + n^{fd}_{sw,i\,|f} v^{fd}_{addr,i\,|f} t^{fd}_{i\,|f} \right) + \left(v^{rev}_{data,i\,|o} t^{rev}_{i\,|o} + n^{rev}_{sw,i\,|f} v^{rev}_{data,i\,|f} t^{rev}_{i\,|f} \right) \right\} + (1-\frac{1}{N})\,(L_{addr} + L_{data} - 2)D_{sw} + t_{mem} \tag{7}$$

The equation for $t^{fd}_{i\,|o}$ and $t^{fd}_{i\,|f}$ are given below. The equations for $t^{rev}_{i\,|o}$ and $t^{rev}_{i\,|f}$ are identical except that the summation is over $j \in \{data\}$. Therefore, we do not give those equations here and drop the superscript *fd* below.

$$t_{i\,|o} = D_{sw} + \sum_{j \in \{addr, addr+data\}} \left\{ A_{j,i\,|o} L_j D_{sw} + P_{busy,j,i\,|o} \frac{L_j D_{sw}}{2} \right\} \tag{8a}$$

$$t_{i\,|f} = D_{sw} + \sum_{j \in \{addr, addr+data\}} \left\{ A_{j,i\,|f} L_j D_{sw} + P_{busy,j,i\,|f} \frac{L_j D_{sw}}{2} \right\} \tag{8b}$$

Finally only $A_{j,i\,|o}$, $A_{j,i\,|f}$, $P_{busy,j,i\,|o}$ and $P_{busy,j,i\,|f}$ remain to be calculated.

$$A_{j,i\,|o} = n_{proc,i\,|f} v_{j,i\,|f} \frac{t_{i\,|f} - D_{sw}}{R} \tag{9a}$$

$$A_{j,i\,|f} = (n_{proc,i\,|f} - 1) v_{j,i\,|f} \frac{t_{i\,|f} - D_{sw}}{R} + v_{j,i\,|o} \frac{t_{i\,|o} - D_{sw}}{R} \tag{9b}$$

$$P_{busy,j,i\,|o} = n_{proc,i\,|f} v_{j,i\,|f} \frac{L_j D_{sw}}{R} \tag{10a}$$

$$P_{busy,j,i\,|f} = (n_{proc,i\,|f} - 1) v_{j,i\,|f} \frac{L_j D_{sw}}{R} + v_{j,i\,|o} \frac{L_j D_{sw}}{R} \tag{10b}$$