# An Analytic Study of Dynamic Hardware and Software Cache Coherence Strategies

Harjinder S. Sandhu and Kenneth C. Sevcik

Computer Systems Research Institute
University of Toronto

## Abstract

Dynamic software cache coherence strategies use information about program sharing behaviour to manage caches at run-time and at a granularity defined by the application. The program-level information is obtained through annotations placed into the application by the user or the compiler. The coherence protocols may range from simple static algorithms to dynamic algorithms that use run-time data structures similar to the directories used in hardware strategies. In this paper, we present an analytic study of five dynamic software cache coherence algorithms and compare these to a representative hardware coherence strategy. The analytic model is constructed using four input parameters -- write probability, locality, granularity, and system size -- and solved by analysis of a Markov chain. We show that the fundamental trade-offs between the different hardware and software strategies are captured in this model. The results of the study show that hardware schemes perform better for fine-grained data structures for much of the parameter space that we study. However, for coarse-grained data structures, various software algorithms are dominant over most of the parameter space. Further, hardware strategies are found to be more susceptible to the effects of contention, and also perform worse for the asymmetric workload that we study.

## 1 Introduction

Dealing effectively with memory access latency is one of the primary concerns in the design of scalable shared-memory processors. Processor caches offer a way to significantly reduce this latency, but they also give rise to the problem of cache coherence. A wide variety of solutions to cache coherence have been proposed. These are often broadly classified as hardware and software strategies. Hardware strategies enforce coherence through architectural support, typically by associating directory information with each cache line sized memory block as in Dash [17] and Alewife [6]. Within the scope of software solutions, caches may be managed through the compiler [8][9], the operating system [21], or through run-time libraries as in the Shared Regions approach [19]. There are inherent trade-offs among each of these choices, as illustrated in Figure 1.

In this paper, we use analytic methods to explore the performance of this last class of strategies, which we refer to generally as dynamic software cache coherence strategies. Annotations in the program, inserted either by the compiler or the user, are used to manage caches dynamically through the run-time system. These annotations, similar to those also proposed in a variety of other systems [3][11], explicitly associate synchronization primitives with the data they protect. A variety of static and dynamic software coherence algorithms are presented and compared along with a representative hardware coherence strategy.

The goals of this work are as follows: (a) to identify a family of algorithms for managing caches dynamically through software, (b) to compare these algorithms with one another and with a hardware coherence strategy, and (c) to determine which of the several software or hardware algorithms performs best for various types of sharing patterns. The results of this study can be applied to both coherent and non-coherent architectures. On non-coherent architectures, there is a need for practical and efficient solutions to cache coherence, as static coherence strategies are still limited by the state-of-the-art in compiler technology, and operating system based approaches generally suffer from the problem of false sharing incurred by managing data at a large fixed-sized granularity. On hardware coherent architectures, the best performance may be achieved by mixing software and hardware control over cacheing, allowing the programmer to select a hardware protocol or one of the available software protocols on a data structure to data structure basis.

Earlier studies of hardware and software strategies have assumed a software model in which caches are managed statically through the compiler. Adve *et al* [1] use Mean Value Analysis to compare a static software scheme to a directory-based hardware scheme. Owicki and Agarwal [18] use an analytic model to compare a static software scheme to a bus-based cache coherence protocol. Static coherence strategies are limited by the fact that a program's run-time behaviour cannot always be predicted accurately at compile-time. As such, these strategies tend to over-invalidate data in caches. Cheong and Veidenbaum [8] study the performance of several compiler-based coherence strategies, one of which attempts to overcome the limitations of static program analysis by using run-time hardware which assigns version numbers to variables and indicates when these are out of date. However, the need for specialized hardware support in this case makes this strategy infeasible for the general class of multiprocessors that are currently being built. In contrast, the software strategies studied in this paper assume no specific hardware support beyond the ability to selectively flush and invalidate data from the caches.

To appear in the proceedings of the ACM Sigmetrics and Performance '95 conference, to be held in Ottawa, Canada, in May 1995.

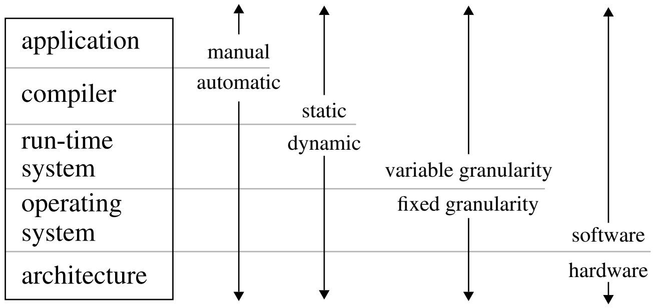



**Figure 1: A classification of cache coherence protocols.**

The primary conclusion of this work is that, given the choice, both hardware and software strategies can play an effective role in managing caches in shared-memory multiprocessors. The analytic model we develop is simple but robust; simple because it uses only four input parameters -- write probability, locality, granularity, and system size --, and robust because it captures the fundamental trade-offs between the different software strategies and the hardware strategy. As expected, for fine-grained data objects, hardware strategies perform better than all of the software strategies for much of the parameter space that we examine. However, for coarse-grained data structures, various software strategies prove to be dominant for most of the parameter space. The hardware strategy is also more susceptible to the problem of contention, and performs significantly worse for single-writer-multiple-reader types of workloads because it tends to generate more cache misses in these cases. Among the dynamic software strategies, the primary trade-off is between writing data from the cache in a pessimistic versus optimistic fashion, with pessimistic writeback strategies performing better when writebacks are necessary more than about 50% of the time.

The organization of this paper is as follows: Section 2 describes a set of algorithms for dynamic software cache coherence. Section 3 presents the analytic model and its solution. Section 4 presents the results of a systematic comparison among the software strategies presented in this paper and the representative hardware strategy. The analytic results are validated against results from a detailed execution-driven simulation study using four applications. The conclusions of this work are presented in Section 5.

# 2 Dynamic software coherence

The Shared Regions (SR) model introduces an abstraction at the program level that explicitly associates synchronization primitives with the data they protect. The term **shared region** is used to denote a set of shared memory locations, not necessarily contiguous, that are bound to a specific synchronization token. The region primitives *readaccess*, *readdone*, and *writeaccess*, *writedone* operate as pairs on a synchronization token and encapsulate a series of accesses to data associated with that token. These region primitives act as mechanisms for both synchronization and run-time coherence management and support a memory consistency framework that has been defined as Entry Consistency [3]. A more detailed discussion of the program-model can be found elsewhere [19].

Software cache coherence strategies usually enforce coherence through memory, using software initiated instructions that write back or invalidate data selectively from the cache. In SR, coherence operations are initiated through software at the *access/done* points in the program, using the following set of instructions as the basic tools:

- *inval*($P_i$): invalidate region from cache *i*
- *wback*($P_i$): write back region from cache *i*
- *flush*($P_i$): invalidate and write back region from cache *i*

These operations operate on a specific region and allow a processor to selectively invalidate or write back data from its own cache, as well as invoke coherence operations on a remote processor's cache. The absence of a parameter in these operations implies a local coherence operation. Algorithms for consistency can be defined based upon *when* these invalidation and writeback operations are invoked. On a particular processor $P_i$, a coherence conflict arises if $P_i$ brings some region into its cache, and that region is modified by some other processor $P_j$ prior to a subsequent reuse by $P_i$. In SR, the completion of the first use of the region $R$ at $P_i$ is indicated by a *done(R)* operation, and the subsequent reuse at $P_i$ is preceded by an *access(R)* operation. Here, an *access(R)* implies either *readaccess(R)* or *writeaccess(R)*, and a *done(R)* implies either *readdone(R)* or *writedone(R)*. The intervening accesses by $P_j$ must necessarily be encapsulated by *writeaccess(R)* and *writedone(R)* operations.

In order to correctly resolve the coherence conflict, two conditions must be satisfied prior to the completion of the *access(R)* at $P_i$: (1) invalidate a stale version of $R$ in $P_i$'s cache, and (2) writeback a dirty version of $R$ from $P_j$'s cache to memory. Each of these actions can be performed at several different points. The invalidation of $R$ at $P_i$ can be performed **pessimistically** when the $P_i$*:done(R)* is invoked, **optimistically** at $P_j$*:{writeaccess(R)*$\rightarrow$ *writedone(R)}*, or in a **lazy** manner at $P_i$*:access(R)*. The write back of $R$ at $P_j$ can be performed **pessimistically** at $P_j$*:writedone(R)* or **optimistically** at $P_i$*:access(R)*. The combination of these alternatives gives rise to six different SR algorithms, one of which is redundant (a pessimistic invalidation of region $R$ in $P_i$ implies a pessimistic writeback of $R$ in $P_j$ as well). These algorithms are labelled *SRp*, *SRlp*, *SRlo*, *SRop*, and *SRoo* and are illustrated in Table 1. The first letter following *SR* indicates the invalidation strategy and the second, when it is present, indicates the writeback strategy. These are chosen from the set {*p*=pessimistic, *o*=optimistic, *l*=lazy, *=any}. The * is useful for describing a group of strategies. For example, *SR*p* denotes all strategies using pessimistic writeback, and *SRl** denotes all strategies using lazy invalidation.

All of the algorithms except *SRp* require run-time state information. For *SRl** and *SRo**, we use a region state vector **s** associated with each region, having one two bit entry per cache in the system. Each entry $\mathbf{s}_i$ describes the state of cache *i* with respect to that region and has three possible states: (1) *Not Have (NH)* - $P_i$ does not have a copy of this region, (2) *Have Valid (HV)* - $P_i$ has a valid copy of this region, and (3) *Have Invalid (HI)* - $P_i$ has an invalid copy of this region. For *SR*o*, a variable $d$ of size $log_2P$ is used to keep track of which cache, if any, has a dirty copy of the

| | **region operation** | ***SRp*** | ***SRlp*** | ***SRop*** | ***SRlo*** | ***SRoo*** |
|---|---|---|---|---|---|---|
| $P_i$ | ↓ *done* | *inval* | | | | |
| $P_j$ | ↓ *waccess* *wdone* | *wback* | *wback* | *inval*($P_i$) *wback* | | *inval*($P_i$) |
| $P_i$ | ↓ *access* | | *inval* | | *inval* *wback*($P_j$) | *wback*($P_j$) |

**Table 1: A family of SR cache coherence algorithms.**

region (with $d$ = -1 implying that no processor has a dirty copy). We assume that the data structure containing both **s** and $d$ is stored in memory and remains uncacheable in order to ensure that it remains consistent. These algorithms are defined in Table 2.

| **Proc *i*** | ***SRp*** | ***SRl**** | ***SRo**** | ***SR*p*** | ***SR*o*** |
|---|---|---|---|---|---|
| *init* | | **for** k=0..$P$ $\mathbf{s}_k = NH$ | **for** k=0..$P$ $\mathbf{s}_k = NH$ | | $d$ = -1 |
| *raccess* *waccess* | | if $\mathbf{s}_i$=$HI$ ***inval*** $\mathbf{s}_i = HV$ | $\mathbf{s}_i$ = HV | | if $d$>=0 and $d$!=$i$ ***wback***($P_d$) $d$ = -1 |
| *rdone* | ***inval*** | | | | |
| *wdone* | ***flush*** | for $\mathbf{s}_j$=$HV$ $\mathbf{s}_j = HI$ | for $\mathbf{s}_j$=$HV$ ***inval***($P_j$) | ***wback*** | $d = i$ |

**Table 2: Implementation of the SR algorithms.**

# 3 Analytic model

In this section, we present an analytic model that is used to study the performance of these software cache coherence algorithms and a directory-based hardware strategy. The hardware directory strategy that is used for comparison is labelled *Dir,* and is assumed to behave like the fullmap directory strategy [5][17]. In this scheme, each cache line sized block in memory has an associated directory which keeps track of which caches have copies of this block. On a cache miss, the directory in memory is consulted and, if other processors have clean copies of the block, they are sent invalidation messages from the memory controller. If another processor has a dirty copy of the block, a writeback message is sent to that processor. For the purpose of this study, we assume an architecture with infinite capacity caches and a cache-line size of one word.

The model is constructed using four input parameters: **write probability *w*, locality *l*, granularity *S*, and system size *N*.** Performance is modelled relative to a single data object of size equal to $S$ cache lines, treated as a shared region in the SR approach and as $S$ independent cache lines by the hardware strategy. The $N$ processors each perform a sequence of reads and writes to this region, where reads and writes are defined as a sequence of operations as follows: In SR, a *read* is a *readaccess* followed by a series of references to the region and terminated by a *readdone* operation, while a *write* is a *writeaccess* followed by a series of updates to each word in the region and terminated by a *writedone* operation. In the hardware scheme, each read and write is just a series of references to each word in the region. Accesses are performed sequentially, and the model does not consider the effects of synchronization or contention for that region.

The write probability $w$ is the probability that the next operation by a processor will be a write. The read probability is $r$=1-$w$. If $P_i$ performed the last read or write operation, it will perform the next one with probability $l$, while any other processor will perform the next read or write with probability $(1\text{-}l)/(N\text{-}1)$. Two types of workloads are considered: *symmetric* and *asymmetric*. In the symmetric workload, $w$ and $l$ are identical for all processors. In the asymmetric workload, the values of $w$ and $l$ may be different for each processor. In this paper, we consider only one class of asymmetric workloads, that of single-writer-multiple-readers. Here, $l$ is the same for all processors, but one processor performs writes with probability $w$ while all other processors perform writes with probability 0.

Earlier studies have considered a workload model that partitions shared data into several classes based upon their access patterns. For instance, Weber and Gupta [22] and Adve *et al* [1] describe five classes of objects: passively-shared, mostly-read, frequently read-written, migratory, and synchronization objects. The parameters used in our study are capable of approximating these sharing patterns. Variations between passively-shared objects, mostly-read objects, and frequently read-written objects are captured by variations of $w$ in the range [0,1]. Migratory objects are those in which $l$ is close to 0 and $w$ is relatively high. Synchronization objects are frequently read-written, widely shared objects in which $S$ is relatively small. Many earlier studies of hardware and software coherence strategies have also considered overall system performance with hardware versus software coherence. Modelling the performance of a single data structure in our study is consistent with the notion that a programmer will not be (or should not be) limited to using just one type of coherence mechanism for all applications or even for all data structures within the same application.

In order to compare the different coherence strategies, a number of parameters need to be derived. These parameters, summarized in Table 3, describe the frequency of cache misses and coherence related operations for each of the coherence algorithms. Cache misses occur only as a result of coherence operations, since infinite caches and a system in steady state are assumed. Invalidations occur whenever clean data is removed from the cache. Writebacks occur when dirty data is ejected from the cache. For SR, the frequency of *access/done* operations and the frequency with which the region status vector is consulted are also derived in order to account for their overhead.Once these parameters have been derived for each algorithm, we use the cost factors (also shown in Table 3) to obtain a measure of the performance of the algorithm. Costs factors are estimated based both upon an implementation of *SRp* and *SRlp* on the Hector multiprocessor [19] and expected average costs on Hector's successor, the NUMAchine architecture. However, the model itself is general enough to be applied to any shared-

| Algorithm-dependent derived parameters | | | Algorithm-independent costs | |
|---|---|---|---|---|
| $M$ | cache misses | frequency of coherence related cache misses | $m$ | memory access - 50 cycles/line |
| $I_s$ | self invalidations | frequency of invalidations on a processor’s own cache, including invalidations initiated by another processor | $i$ | invalidation - 2 cycles/line |
| $I_r$ | remote invalidations | frequency of invalidations sent to a remote site | $g$ | messages - 60 cycles |
| $E_s$ | self writebacks | frequency of writebacks on a processor’s own cache, including writebacks initiated by some other processor | $e$ | writeback - 50 cycles/line |
| $E_r$ | remote writebacks | frequency of writebacks initiated to a remote site | $c$ | context switch - 8 cycles |
| $B$ | *access* operations | for SR, frequency of readaccess/writeaccess operations | $b$ | SR base overhead - 200 cycles |
| $V$ | region status checks | for SR, frequency of check/update region status vector | $v$ | SR check status - 60 cycles |

**Table 3: Algorithm dependent derived parameters and algorithm independent costs.**

memory multiprocessor. In the next subsection, we describe how these parameters are derived for each coherence algorithm based upon the four input parameters.

## 3.1 Model construction

To determine the frequency with which cache misses and coherence related operations occur, we construct a Markov process that describes the behaviour of the system in terms of the input parameters. Each state describes the status of a single region with respect to all of the processors in the system. Transitions represent *read* and *write* operations to this region, using the definition of reads and writes provided earlier, as a *series* of references to the region, and encapsulated by access/done operations in the SR version. We describe the symmetric two processor system in detail first.

The model is illustrated in Figure 2. One processor is represented as $P_1$, and the other as $P_2$. There are four states: *01, 10, 00a, 00b*. In state *01*, processor $P_1$ has a valid (dirty) copy of the region within its cache but $P_2$ does not, while in state *10*, $P_2$ has a valid copy of the region but $P_1$ does not. In states *00a* and *00b*, both $P_1$ and $P_2$ have valid copies of the region. In state *00a*, $P_1$ has performed the most recent read of the region, while in state *00b*, $P_2$ is the most recent reader. Reads and writes by $P_1$ are labelled $R_1$ and $W_1$, while reads and writes by $P_2$ are labelled $R_2$ and $W_2$.

The first step is to calculate the relative frequencies with which each state is visited: $V_{01}$,$V_{10}$,$V_{00a}$, and $V_{00b}$. To do this, the transitions are first labelled according to the probabilities with which they are taken. These transition probabilities are based upon $w$ and $l$ (and $N$ for more than two processors) and are shown in Figure 3. Next, we solve the set of global balance equations that relate the relative visit frequencies of each of the four states. The resulting relative visit frequencies are also shown in Figure 3 within the boxes that represent each state.

The next step is to determine the frequency of cache misses and coherence related operations. This is done by mapping the semantics of each coherence algorithm (as described in Table 2) onto the Markov model of Figure 2. This mapping is shown in Table 4. Here, for each algorithm, we show which transitions in the Markov chain correspond

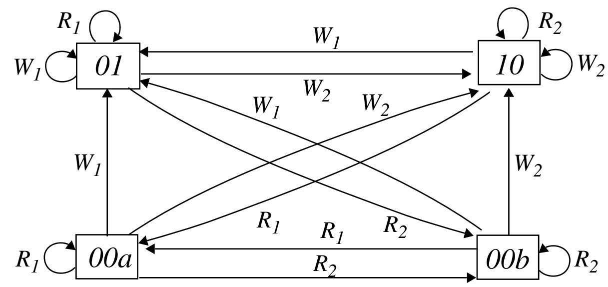



**Figure 2: Symmetric two-processor Markov model.**

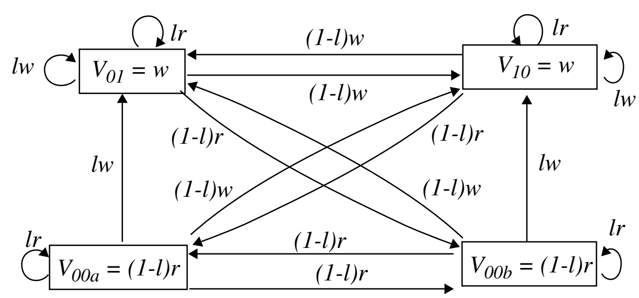



**Figure 3: Symmetric two-processor model solution.**

to cache misses, invalidations, writebacks, and region status vector checks. These are shown with respect to $P_1$ only, so invalidations and writebacks imply operations on $P_1$’s cache. As an example of how entries in this table are obtained, consider cache misses in *SRlp*. In *SRlp*, $P_1$ accesses the region from memory whenever it performs a read or write from a state in which $P_1$ does not have a valid copy of the region. In the Markov process, this corresponds to transitions $R_1$ and $W_1$ from state *10*.

Once this mapping has been created, we derive the frequency with which operations are performed by taking the sum of the transitions on which those operations occur. For example, in *SRlp*, caches misses in $P_1$ occur on transitions $R_1$ and $W_1$ from state *10*. The frequency is given by $V_{10}(R_1+W_1) = w[(1\text{-}l)r+(1\text{-}l)w] = w(1\text{-}l)$. All of the derived parameters, summarized in Table 5, are obtained in a similar manner. Details of the derivation are given in Appendix A.

Next, the frequency of cache misses and coherence operations is combined with the cost model shown in Table 3 to

| Alg | cache miss | invalidations | write-backs | vector check |
|---|---|---|---|---|
| *SRp* | *All:* $R_1$,$W_1$ | *All:*$R_1$ | *All:*$W_1$ | |
| *SRlp* | *10:*$R_1$,$W_1$ | *10:*$R_1$,$W_1$ | *All:*$W_1$ | *All:*$R_1$,$W_1$ |
| *SRop* | *10:*$R_1$,$W_1$ | *01,00a,00b:*$W_2$ | *All:*$W_1$ | *01,00a,00b:*$W_2$ |
| *SRlo* | *10:*$R_1$,$W_1$ | *10:*$R_1$,$W_1$ | *01:*$R_2$,$W_2$ | *All:*$R_1$,$W_1$ |
| *SRoo* | *10:*$R_1$,$W_1$ | *00a,00b:*$W_2$ | *01:*$R_2$,$W_2$ | *01,00a,00b:*$W_2$ |
| *Dir* | *10:*$R_1$,$W_1$+ *00a,00b:*$W_1$ | *00a,00b:*$W_2$ | *01:*$R_2$,$W_2$ | |

**Table 4: Mapping the algorithm semantics onto the Markov chain. Properties are with respect to $P_1$ only. Entries $s_1$,$s_2$:*a,b* imply transitions *a* and *b* from state $s_1$ and $s_2$. *All*=All states.**

| Alg | $M$ | $I_s$ | $I_r$ | $E_s$ | $E_s$ | $B$ | $V$ |
|---|---|---|---|---|---|---|---|
| *SRp* | (1-*rl*) | *r*(1-*rl*) | | *w*(1-*rl*) | | (1-*rl*) | |
| *SRlp* | *w*(1-*l*) | *w*(1-*l*) | | *w*(1-*rl*) | | (1-*rl*) | (1-*rl*) |
| *SRop* | *w*(1-*l*) | *w*(1-*l*) | *w*(1-*l*) | *w*(1-*rl*) | | (1-*rl*) | *w*(1-*l*) |
| *SRlo* | *w*(1-*l*) | *w*(1-*l*) | | *w*(1-*l*) | *w*(1-*l*) | (1-*rl*) | (1-*rl*) |
| *SRoo* | *w*(1-*l*) | *rw*(1-*l*) | *rw*(1-*l*) | *w*(1-*l*) | *w*(1-*l*) | (1-*rl*) | *w*(1-*l*) |
| *Dir* | *w*(1-*l*)(1+*r*) | | *rw*(1-*l*) | | *w*(1-*l*) | | |

**Table 5: Relative frequencies for $P_1$ in the symmetric system.**

obtain an overall estimate of the performance of each algorithm. This procedure is described below.

**Cache misses**: The cost of cache misses for each algorithm is obtained by multiplying *M*, the frequency with which cache misses occur, with *S* times *m*, the cost of accessing the region from memory. *SRp* has the most cache misses because of its pessimistic invalidation behaviour. *Dir* incurs more cache misses than the dynamic SR algorithms because in *Dir*, writes by $P_1$ from a state in which $P_2$ also has a valid copy of the region result in a cache miss at $P_1$.

**SR overhead**: In all of the SR algorithms, each *access* operation incurs with frequency *B* a basic overhead cost *b*. In *SRl**, there is an additional cost *v* on each *access* to check the cache state vector. In *SRo**, the cache status vector is only consulted when remote writebacks or invalidations are required.

**Writebacks**: The cost of writebacks for the pessimistic writeback strategies *SR*p* is $E_s eS$. In the optimistic writeback strategies *SR*o*, the cost to $P_1$ for writebacks initiated by $P_2$ is $E_s(2c+eS)$. This takes into account that $P_1$ must perform context switches in order to handle a remote writeback request. When $P_1$ initiates writebacks to $P_2$, $P_1$ sends a message to $P_2$ and then stalls. When $P_2$ receives the message, it context switches, performs the writeback and then sends a message back to $P_1$. The cost to $P_1$ is then $E_r(2g+c+eS)$. For *Dir*, we assume that cache line writebacks initiated from $P_2$ to $P_1$ do not incur a cost at $P_1$, since such requests can be processed at a lower level within the hardware and do not need to interrupt $P_1$. We assume the cost of a single write-back for the initiating processor to be similar to the cost of writing back a single line in SR, an assumption that favours the *Dir* strategy because writing a large set of lines as in SR can be made more efficient, possibly by avoiding per-line acknowledgments or by using *bulk* data transfers.

**Invalidations:** The cost of invalidations in *SRp*, *SRlp*, and *SRlo* is $I_s iS$. In *SRo**, the cost to $P_1$ for invalidations initiated by $P_2$ is $I_s(2c+iS)$. For invalidations from $P_1$ to $P_2$, $P_1$ can initiate the invalidation but continue processing rather than blocking, and the completion of outstanding invalidation requests can be checked at a later time. Then, the cost to $P_1$ for initiating invalidation requests is just the cost of composing a message. For the hardware strategy, we assume the behaviour of invalidation requests to be similar to writeback operations.

## 3.2 Increasing system size

For the symmetric *N* processor system, the same Markov chain as shown in Figure 2 is used, but the intepretation of the states is changed so that states *01* and *00a* reflect the state of one processor, while states *10* and *00b* reflect the state of the other *N*-1 processors in the system. This system, and its solution, is shown in Appendix B.

When remote invalidations are sent in *SRop*, *SRoo*, and *Dir* in the *N* processor system, more than one invalidation message may have to be sent. Multiple invalidations do not affect latency directly because they can proceed simultaneously. However, when we measure the effects of contention, the traffic generated by multiple invalidations may play a significant role.

In the symmetric system, the problem of determining how many invalidation messages are sent when invalidation is required can be simplified to determining how many unique processors will perform read operations on average in between any two writes in the system. To do this, we set up and solve another Markov chain shown in Figure 4.

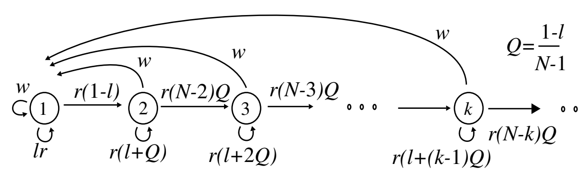



**Figure 4: A Markov chain to determine the average number of processors that have valid copies of a region whenever a write operation occurs in the symmetric system.**

Here, each state indicates how many unique processors have valid copies of the region. Any write takes the system back to state 1. From state 1, a local read will keep the system in state 1, while a remote read will take it to state 2, indicating that two processors now have valid copies of the region. From state 2, a read by either of the two processors that have already acquired copies of the region will keep the system in this state. The processor that performed the last read operation does so again with probability *lr*, while the other processor with a valid copy of the region performs a read with probability *Qr*, where *Q* is the probability that a specific non-local processor performs the next operation, and is equal to (1-$l$)/$N$-1. A read operation from any of the other $N$-2 processors takes the system to state 3, and has probability ($N$-2)$Qr$. (Note that $1-(l+(k-1)Q)$ is equal to $(N-k)Q$ since $l+(N-1)Q = 1$). The system arrives at state $k$ from state $k$-1 with probability $(N-(k-1))Qr$, and stays in state $k$ with probability $l+(k-1)Q$. The equation for $V_1$ is solved using the assumption that $\Sigma V_i = 1$:

$$V_1 = (rl+w)V_1 + w(1-V_1) = \frac{w}{1-rl}$$

Next, since the $k$-$th$ state is dependent only on the ($k$-1)-$th$ state, the equation for state $k$ can be written first in terms of the ($k$-1)-$th$ state, and then transformed into an equation that is a function of $V_1$ only.

$$V_k = \frac{rQ(N-k+1)}{w+r(N-k)Q}V_{k-1} = V_1\prod_{j=1}^{k-1}\frac{rQ(N-j)}{w+rQ(N-j-1)}$$

In state $V_i$, $i$ processors have valid copies of the region. A write transition from state $i$ requires ($i$-1) invalidations if the write is from one of the $i$ processors that already have copies of the region, and $i$ invalidations otherwise. The average number of invalidations $N_{inv}$ on a write transition is calculated in the following way:

$$N_{inv} = \sum_{i=1}^{N}\left((i-1)V_i(l+(i-1)Q) + iV_i(N-i)Q\right)$$

## 3.3 Asymmetric system

In the last subsection, we have described in detail how the symmetric model is constructed and solved. The asymmetric model is constructed and solved in a similar fashion except that the Markov chain has only three states *01*, *00a*, and *00b*. Taking $P_1$ to be the only writer, state *10* is absent because $P_2$ never performs a write operation. Further, unlike the symmetric case in which performance is measured relative to $P_1$, the average performance of all processors is considered in the asymmetric system.

# 4 Results

In this section, we present a comparative analysis of the SR algorithms and the hardware strategy. The results are presented systematically, first for the symmetric two-processor system, and then for $N$ processors and asymmetric system later. In presenting the results, we rely on two types of graphs. For comparisons between any two coherence strategies, we use contour graphs (Figures 5-9). In these, two workload parameters form the $x$ and $y$ axis, and the graph shows the regions in which each algorithm is better than the other using a thick contour line, and the degree by which they are better using thinner contour lines. Each of the thin contour lines represents an additional 10% difference in performance over the previous one. Spectral graphs are used for comparing all of the strategies (Figures 10 and 11). Here, for values of $w$ and $l$ along the $x$ and $y$ axis respectively, the graph shows the area in which each algorithm is better than all of the others.

## 4.1 Symmetric workload

**Pessimistic versus lazy invalidation**

The performance of *SRp* relative to *SRlp* depends on the cost and frequency of unnecessary memory accesses and invalidations. From Table 5, the frequency of memory operations for *SRp* is (1-$rl$), and the frequency of invalidations is $r$(1-$rl$). Since the frequency of invalidations is always less than or equal to the number of memory operations, and the cost is significantly less (two cycles versus 50 cycles), it stands to reason that the cost of the invalidations in *SRp* will never have a significant impact on performance. However, the cost of the additional memory accesses is quite significant, as shown in Figure 5. For any value of $l$, *SRp* is better than *SRlp* when the region size is smaller than three to five cache lines. For the small regions, the cost of checking the region state vector (in memory) upon every access in *SRlp* becomes a significant factor. Under low locality and high write probability, the difference between the number of memory operations in *SRp* and *SRlp* begins to fade. Thus, increases in $w$ have the effect of moving the *SRp* versus *SRlp* contour slightly to the right for low values of $l$, but *SRlp* remains significantly better than *SRp* for the majority of cases.

**Optimistic versus lazy invalidation**

Lazy invalidation strategies incur the cost of checking the region state vector on every *access* operation, while optimistic strategies do not need to consult the region state vector as often. On the other hand, invalidations in the optimistic strategies are initiated from a remote processor, and therefore incur an additional cost in performing context switches and also in composing messages to send. Since all of these costs are relatively small compared to the cost of memory accesses and writeback operations, the differences between *SRop* and *SRlp* are rather small for most values of $l$ and $w$, as shown in Figure 6. *SRlp* performs 0-10% better when memory accesses are performed frequently (for low $l$ and high $w$). When $w$ is very low (less than 0.05), memory access and coherence operations are virtually absent, and the only costs incurred are those of checking the region vector for *SRlp*. In these cases, *SRlp* performs significantly worse than *SRop*.

**Optimistic versus pessimistic writeback**

Intuitively, pessimistic writeback strategies will perform better than optimistic writeback strategies when writebacks are often necessary (for low $l$ and high $w$). The cost of performing writeback operations in the optimistic writeback strategy *SRlo* is just over two times that of *SRlp*, since in *SRlo*, both processors are blocked while remote writebacks occur, and there is an additional cost of sending a message and interrupting a processor. At $w$=1, $l$=0.5, *SRlo* performs half as many writebacks as *SRlp*, but since their cost is just over two times that of *SRlp*, *SRlp* still performs a little better. Thus, the optimistic strategy performs better when the number of writebacks performed is less than half of those performed in the pessimistic strategy. This is shown in Figure 7.

**Hardware versus dynamic software**

The hardware strategy differs from the dynamic software strategies in a number of ways: (1) the frequency of memory operations, (2) the granularity at which data is managed, and

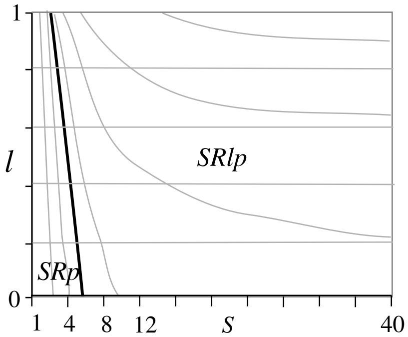


**Figure 5:** ***SRp*** **versus** ***SRlp*****,** ***w*****=0.5,** ***N*****=2.**

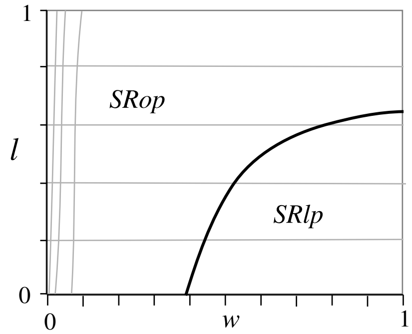


**Figure 6:** ***SRop*** **versus** ***SRlp*****,** ***S*****=256,*****N*****=2.**

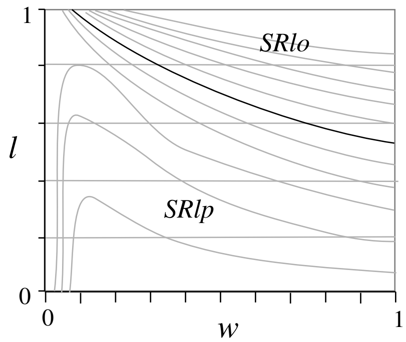


**Figure 7:** ***SRlp*** **versus** ***SRlo,*** ***S*****=256,** ***N*****=2.**

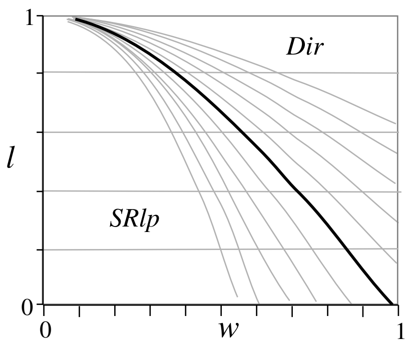


**Figure 8:** ***SRlp*** **versus** ***Dir*****,** ***S*****=256,** ***N*****=2.**

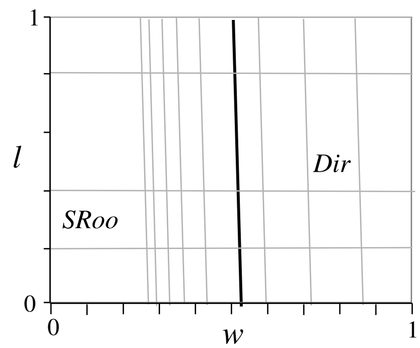


**Figure 9:** ***SRoo*** **versus** ***Dir*****,** ***S*****=256,** ***N*****=2.**

(3) the way in which invalidations and writebacks are handled. For the symmetric workload, the hardware strategy performs *(*1+*r)* times more memory operations than the dynamic SR algorithms (see Table 5). However, as *r* increases, the overall impact of memory operations decreases as the total frequency of memory operations *(*1-*l)w(*1+*r)* becomes smaller.

Figures 8 and 9 compare the hardware strategy to *SRlp* and *SRoo,* respectively. The way in which coherence operations are performed in *Dir* is similar to *SRoo*, with the exception that in *Dir* these operations are sent on a per-line basis and do not interrupt the remote processor. The advantage that the dynamic SR algorithms have over *Dir* is fewer caches misses and more efficient handling of invalidations. In *SRlp*, for instance, aside from the region overhead, invalidations complete entirely within the cache. In *SRoo*, a single remote invalidation request is issued to a remote processor, and the per-line invalidations complete in the remote cache. In *Dir*, invalidations are initiated from memory to a remote processor on a per-line basis.

*SRlp* performs better than *Dir* for low *l*, where the frequency of writebacks is closer to that of *Dir,* and also for lower *w,* where the greater number of memory operations performed in *Dir* have a greater impact. The break-even point comes close to the line defined by *l*+*w*=1. *SRoo* performs better for low values of *w*. The break-even point occurs at close to *w*=0.5. For much smaller regions, the region overhead plays a more significant role and the trade-offs change significantly, as discussed in the next section.

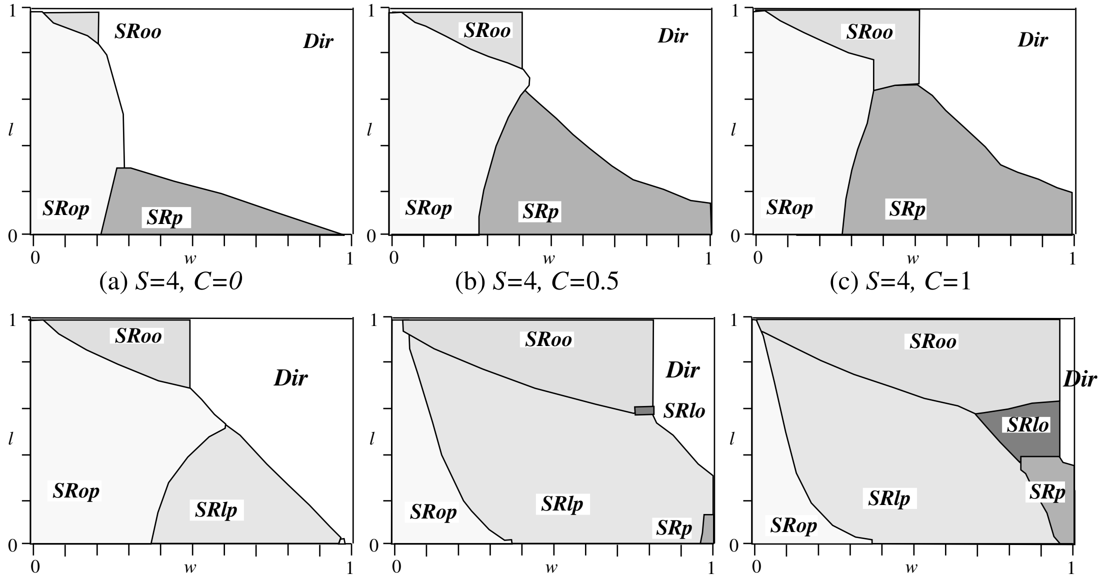
**Overall Comparison**

In this section, we examine all of the strategies together, and try to determine which ones will perform best for specific values of *l*, *w,* and *S*. We also include an estimate of the effects of contention. Since performance is measured relative to a single region in our model, it is impossible to derive a good measure of contention. Our estimate relies on the intuition that the strategies that generate more network traffic will also induce more contention. For each algorithm *A*, we derive the number of network transactions for each algorithm, called $A_{net}$, by replacing cost values in Table 3 with the number of network transactions for each type of operation from or to $P_1$. For all algorithms, the lowest $A_{net}$ for specific values of *w*, *l*, *S*, and *N* is called the *base.* The contention experienced by algorithm *A* is then,

$$\frac{base + (A_{net} - base)\,C}{base}$$

where *C* is the contention expansion factor. For *C*=0, an algorithm experiences no contention. For *C*>=0, algorithm *A* experiences contention directly proportional to the number of network transactions it generates relative to the strategy which generates the least number of transactions. Note that this is independent both of the number of processors and other activity in the system (unrelated to the region we model), so that in practice the number of message delays due to contention could be higher than *C*=1.

Figure 10 shows the comparative results for all of the coherence algorithms for *S*=4 and *S*=256, and for various values of *C*. First, let's examine the results for *C*=0, where there is no contention. For *S*=256 (Figure 10(d)), the results follow directly from the comparisons presented in the earlier subsections. Indeed, this figure could have been constructed by overlaying the earlier graphs (Figures 5 to 9) to produce a single image. The hardware strategy is best for higher values of *w* and *l,* and the pessimistic writeback algorithms *SRlp* and *SRlo* are best for lower values of *w* and *l*, as before. *SRoo*




(a) $S$=4, $C$=0 (b) $S$=4, $C$=0.5 (c) $S$=4, $C$=1

(d) $S$=256, $C$=0 (e) $S$=256, $C$=0.5 (f) $S$=256, $C$=1

**Figure 10: Results for the symmetric two processor workload.**

occupies a small region for $l$ close to 1 and $w$ less than 0.5. However, all of the dynamic SR algorithms suffer when the region is much smaller, as seen in Figure 10(a). Here, *Dir* is dominant over a much larger area, and *SRlp* is replaced with *SRp*.

When contention is added to the system, the results begin to change significantly. *Dir* generates the most network traffic and is therefore affected the most. This is especially apparent for $S$=256, where the area occupied by *Dir* shrinks considerably from $C$=0 to $C$=1. For $C$=1, *Dir* is best only for $w$>.98, and $l$>.4. *SRlp*, *SRlo*, and *SRoo* take over most of the area formerly occupied by *Dir* for increasing values of $C$. *SRlp* also begins to take over much of the area occupied by *SRop*. *SRp* and *SRlo,* which were never best for any combination of $l$ and $w$ for $C$=0, occupy a portion of the space for large regions and $C$=1. Under these conditions ($S$=256, $C$=1), each of the six algorithms we have studied proves to be best for a significant portion of the $(l,w)$ plane. For $S$=4, the relative effects of contention are less, as the region size is closer to the granularity at which data is managed in *Dir.*

## 4.2 Asymmetric workload

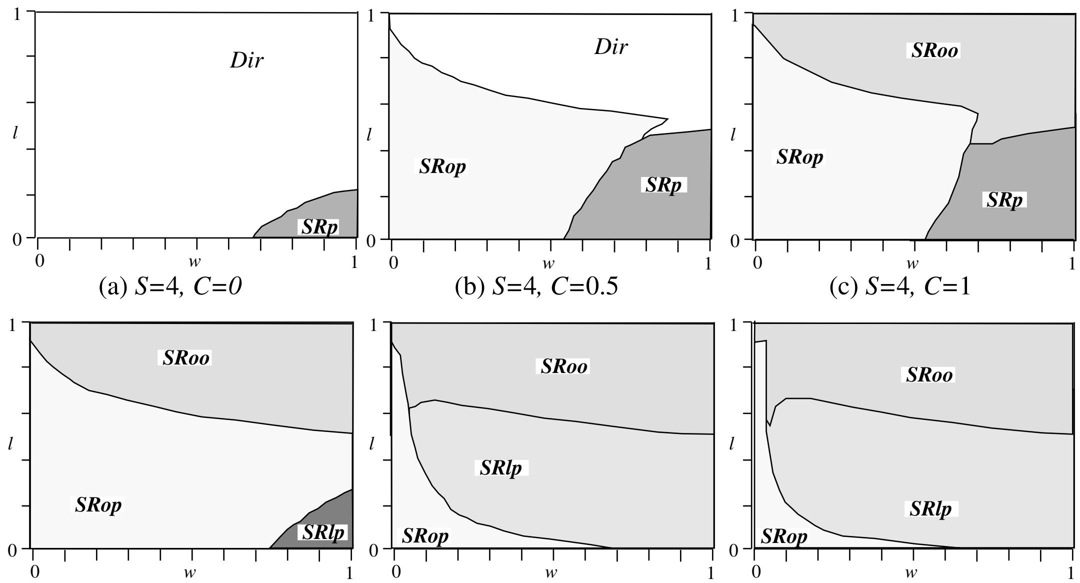
Results for the asymmetric workload, shown in Figure 11, are significantly different than those for the symmetric workload. For $S$=4 and $C$=0, the *Dir* strategy is best for all but a small area occupied by *SRp* and defined by high values of $w$ and low values of $l$. Yet, for larger regions, or for $C$=1 for smaller regions, *Dir* is entirely absent. The reason for this behaviour is due to the frequency of memory operations for *Dir* in the single-writer-multiple-reader case. Since $P_1$ is the only processor that modifies the region, the copy of the region in $P_1$'s cache is always valid. In the dynamic SR algorithms, $P_1$ always accesses the region from cache. In *Dir*, every write by $P_1$ from a state in which $P_2$ also has a valid copy of the region results in memory operations by $P_1$. In this system, the number of memory operations performed is taken as the average of both $P_1$ and $P_2$, and the dynamic SR algorithms perform memory operations with frequency $w(1-l)$, while the hardware strategy does so with frequency $2w(1-l)$. This accounts for the poor performance of *Dir* in the asymmetric system. The trade-off boundaries between *SRoo* and *SRop* also shift in favour of *SRoo*. This is a result of combining results for $P_2$, which never performs any writes, with results for $P_1$, which performs writes with probability $w$ and always has valid copies of the data in its own cache.

## 4.3 System size

The impact of increasing the number of processors is more contention in the system. In the optimistic invalidation algorithms *SRop*, *SRoo*, and in *Dir*, the average number of invalidation messages that are sent will increase, but the effect is only one of increased system traffic and contention as multiple invalidation requests can take place simultaneously. The results, although not shown in the interest of brevity, are similar to the two-processor cases. The effects of contention are similar but significantly more pronounced for $C$>0 as $N$ is increased. That is, the *Dir* strategy suffers most from increases in system size, followed by the optimistic invalidation strategies *SRop* and *SRoo*.

## 4.4 Validation

The coherence algorithms presented in this paper were implemented in a detailed execution-driven multiprocessor simulator and studied using a set of coarse-grained numerical applications. The results of that simulation study are presented separately [20]. Here, we compare the simulation results for four coarse-grained applications with results from the analytic model presented in this paper. These applications are: Fast Fourier Transform (FFT), LU-Decomposition (LU), Mean-Value Analysis (MVA), and Neural Networks (NRL). SR annotations were inserted into each application.




(a) $S$=4, $C$=0 (b) $S$=4, $C$=0.5 (c) $S$=4, $C$=1

(d) $S$=256, $C$=0 (e) $S$=256, $C$=0.5 (f) $S$=256, $C$=1

**Figure 11: Results for the asymmetric single-writer-multiple-reader workload, with two-processors.**

The simulator is patterned after the Hector and NUMAchine architectures [21]. A hardware coherence strategy similar to the one described in this paper was also implemented in the simulator. Figure 12 shows the relative performance of the six coherence algorithms for each of the four applications. Arrows separate algorithms for which distinctly different levels of performance are observed or predicted. For the simulator, a difference of 5% or more in the run-time of the application is considered significant. For the analytic model, 10% or more is considered significant.

The results from the analytic model are very close to that of the simulation. In two of the applications, MVA and LU, simulation results are exactly as predicted by the analytic model. For FFT and NRL, the hardware scheme (*Dir*) appears better in the analysis that it is observed to be under simulation. In FFT, differences arise from the fact that loads followed by stores to the same address and from the same processor generate cache misses that were not expected in the analytic model. In NRL, the analytic results predict that *Dir* will perform better than all of the other strategies, but in the simulation results, there is no significant difference between *SRoo*, *SRlo*, and *Dir*. The reason for this is that when $l$ is close to 1, very little coherence traffic is being generated for the optimistic writeback algorithms and for *Dir*, so that any difference in the behaviour of these algorithms is not visible in the overall run-time of the program.

One detail that was not considered in the analytic results presented here is the effect of architectural variations. In the simulation results however, a number of architectural features are varied, including: cache size, network latency, and the use of store buffers and weak consistency [20]. Although a detailed discussion of their effects is beyond the scope of this paper, in general, the ordering of algorithms shown in Figure 12 remains unchanged for a wide range of architectural variations. This provides a degree of confidence that the same will be true in the analytic results as well.

# 5 Conclusions

In this paper, we have presented a family of algorithms for managing caches dynamically through software. An analytic model was constructed to compare these software algorithms and a representative hardware strategy. We find that while the hardware strategy is better than the software algorithms for most of the parameter space for fine-grained data, various software algorithms are dominant for most of the parameter space for coarse-grained data, particularly when contention is incorporated into the model. Further, the hardware strategies perform worse for the asymmetric workloads studied here because they tend to incur more cache misses in these cases. The analytic predictions are compared for four applications to the results of a detailed simulation study and found to be fairly consistent, although *Dir* seems to be favoured more in the analysis than it is in the simulation.

The work presented in this paper represents a departure from the analysis presented in earlier comparative studies of hardware and software cache coherence strategies. Much of this earlier work has studied static software coherence strategies, rather than the dynamic software approaches considered here. Further, much of this earlier work has been concerned with evaluating overall system performance using a hardware or software strategy. In practice, the choice of whether or not to implement cache coherence in hardware is based less upon the performance of these hardware protocols relative to the software strategies, and more on factors such as the cost and complexity of implementing coherence in hardware. For non-coherent architectures, the strategies we have studied in this paper present a practical solution to the problem of cache coherence. For coherent architectures, we have shown that the selective use of software coherence protocols when it is practical and efficient to do so may result in the best overall system performance.

| **FFT**<br>**symmetric, *w*=1, *l*=.6** | | **LU**<br>**single-writer, *w*=1, *l*=0** | | **MVA**<br>**single-writer, *w*=.5, *l*=0** | | **NRL**<br>**symmetric, *w*=1, *l*=1** | |
|---|---|---|---|---|---|---|---|
| **simulation** | **analytic** | **simulation** | **analytic** | **simualtion** | **analytic** | **simulation** | **analytic** |
| *SRoo*<br>*SRlo*<br>↓<br>*Dir*<br>↓<br>*SRlp*<br>*SRop*<br>↓<br>*SRp* | *Dir*<br>↓<br>*SRoo*<br>*SRlo*<br>↓<br>*SRlp*<br>*SRop*<br>↓<br>*SRp* | *SRlp*<br>*SRop*<br>↓<br>*SRp*<br>*SRoo*<br>*SRlo*<br>↓<br>*Dir* | *SRlp*<br>*SRop*<br>↓<br>*SRp*<br>*SRoo*<br>*SRlo*<br>↓<br>*Dir* | *SRop*<br>*SRlp*<br>↓<br>*SRoo*<br>*SRlo*<br>↓<br>*SRp*<br>↓<br>*Dir* | *SRop*<br>*SRlp*<br>↓<br>*SRoo*<br>*SRlo*<br>↓<br>*SRp*<br>↓<br>*Dir* | *SRlo*<br>*SRoo*<br>*Dir*<br>↓<br>*SRlp*<br>*SRop*<br>↓<br>*SRp* | *Dir*<br>↓<br>*SRlo*<br>*SRoo*<br>↓<br>*SRlp*<br>*SRop*<br>↓<br>*SRp* |

**Figure 12: Simulation results versus analytic predictions. Algorithms are ranked best to worst.**


## Acknowledgements

We would like to thank the referees for their many useful comments on this paper, and also thanks to Xiaohan Qin of the University of Washington and one of the referees for detecting an error in the symmetric *N* processor Markov system in an earlier version of this paper.


# References


[1] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of hardware and software cache coherence strategies. In *18th Int'l. Symp. on Computer Architecture*, pages 298-307, May 1991.

[2] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy. An evaluation of directory schemes for cache coherence. In *15th Int'l. Symp. on Computer Architecture*, pages 280-289, Jun 1988.

[3] B. Bershad, M. Zekauskas and W. Sawdon. The Midway distributed shared memory system, In *Proc. of COMPCOM'93*, pages 528-537, Feb 1993.

[4] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *13th Symp. on Operating Systems Principles*, pages 152-164, Oct 1991.

[5] L. Censier and P. Feautrier, A new solution to coherence problems in multicache systems. In *IEEE Transactions on Computers*, c27(12), pages 1112-1118, Dec 1978.

[6] D. Chaiken, J. Kubiatowicz and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme, *Fourth Int'l. Symp. on Architectural Support for Programming Languages and Operating Systems*, Apr 1991.

[7] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *Eighth Int'l. Conference on Supercomputing*, Jul 1994.

[8] H. Cheong and A. Veidenbaum. Compiler-directed cache management for multiprocessors. *IEEE Computer*. 23(6), pages 39-47, Jun 1990.

[9] R. Cytron, S. Karlovsky, and K. McAuliffe. Automatic management of programmable caches. In *Proc. of the Int'l. Conference on Parallel Processing*, Aug 1988.

[10] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *15th Int'l. Symp. on Computer Architecture*, May 1988.

[11] M. Feeley and H. Levy. Distributed shared memory with versioned objects. In *Conf. on Object-Oriented Programming Systems Languages, and Applications*, Oct 1992.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennesy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *16th Int'l. Symp. on Computer Architecture*, May 1990.

[13] M. Hill, J. Larus and S. Reinhardt and D. Wood. Cooperative Shared-Memory: Software and hardware support for scalable multiprocessors. In *Fifth Int'l. Symp. on Architectural Support for Programming Languages and Operating Systems*, Oct 1992.

[14] P. Keleher, A. Cox and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *18th Int'l. Symp. on Computer Architecture*, May 1992

[15] M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *3rd Symp. on Principles and Practices of Parallel Programming*, Apr 1991.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*. 28(9),pp.690-691, Sep 1979.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th Int'l. Symp. on Computer Architecture*, pages 148-159, May 1990.

[18] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherency. In *Proc. 3rd Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr 1989.

[19] H. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions approach to software cache coherence. In *Symp. on Principles and Practices of Parallel Programming*,May 1993.

[20] H. Sandhu. Shared Regions: A strategy for efficient cache coherence on shared-memory multiprocessors. Ph.D. Thesis, University of Toronto, In preparation.

[21] Z.G. Vranesic, M. Stumm, D. Lewis and R. White. Hector - a hierarchically structured shared-memory multiprocessor, In *IEEE Computer*, Jan 1991.

[22] W. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr 1989.

# Appendix A

Derivations of the frequency of cache misses and coherence operations for $P_1$ in the symmetric system for $N$=2 are shown here (derivations for the $N$>2 system are similar). These are derived by mapping the semantics of each coherence algorithm onto the Markov chain shown in Figure 2. This mapping is shown for each algorithm in Table 4. The symbols $M$, $I_s$, $I_r$, $E_s$, and $E_r$ are defined in Table 3. To understand the semantics of each algorithm, refer to Table 2. The visit frequencies for each state and the transition probabilities are given in Figure 3.

**SRp:**

The region is accessed from memory on every access operation. Then, to derive the frequency of cache misses, we take the sum of all transitions by $P_1$. Invalidations are performed pessimistically on every *readdone* operation, which occurs on every read transition by $P_1$. Writebacks are performed pessimistically on every *writedone* operation, which occurs on every write transition by $P_1$.

$$M = V_{01}[R_1+W_1]+V_{10}[R_1+W_1]+V_{00a}[R_1+W_1]+V_{00b}[R_1+W_1]$$
$$= lw+w[(1-l)w+(1-l)r]+(1-l)lr+(1-l)r[(1-l)w+(1-l)r]$$
$$= wl+w(1-l)+(1-l)rl+ (1-lr)(1-l) = w+(1- l)r = 1- rl$$
$$Is = V_{01}[R_1]+V_{10}[R_1]+V_{00a}[R_1]+V_{00b}[R_1]$$
$$= w[lr]+w[(1-l)r]+(1-l)r[lr]+(1-l)r[(1-l)r]$$
$$= wr(l+1-l)+(1-l)r[lr+r-rl] = wr+(1-l)rr = r(1-rl)$$
$$Es = V_{01}[W_1]+V_{10}[W_1]+V_{00a}[W_1]+V_{00b}[W_1]$$
$$= w[lw]+w[(1-l)w]+(1-l)r[lw]+(1-l)r[(1-l)w]$$
$$= w[lw+w-lw]+(1-l)r[lw+w-lw] = w[w+(1-r)r] = w(1-rl)$$

**SRlp:**

Cache misses occur on operations by $P_1$ from a state in which $P_1$ does not have a valid copy of the region, transitions $R_1$ and $W_1$ from state *10*. Invalidations are performed on these same transitions, since state *10* indicates that $P_1$ currently has an invalid copy of the region and must invalidate its own copy on a $R_1$ or $W_1$ transition. Writebacks are performed pessimistically as in *SRp*.

$$M = V_{10}[R_1+W_1] = w[(1-l)r+(1-l)w] = w(1-l)$$
$$Is = V_{10}[R_1+W_1] = w(1-l)$$
$$Es = V_{01}[W_1]+V_{10}[W_1]+V_{00a}[W_1]+V_{00b}[W_1] = w(1-rl)$$

**SRop:**

Caches misses occur as in *SRlp*. Invalidations at $P_1$ are initiated by $P_2$ whenever $P_2$ performs a write from a state in which $P_1$ currently has a valid copy of the region, states $V_{10}$, $V_{00a}$, and $V_{00b}$. Since the system is symmetric, $I_s$=$I_r$. Writebacks are as in *SRlp*.

$$M = V_{10}[R_1+W_1] = w(1-l)$$
$$Is = V_{01}[W_2]+V_{00a}[W_2]+V_{00b}[W_2]$$
$$= w(1-l)w+(1-l)r(1-l)w+(1-l)rlw = w(1-l)$$
$$Ir = V_{10}[W_1]+V_{00a}[W_1]+V_{00b}[W_1]$$
$$= w(1-l)w+(1-l)r(1-l)w+(1-l)rlw = w(1-l)$$
$$Es = V_{01}[W_1]+V_{10}[W_1]+V_{00a}[W_1]+V_{00b}[W_1] = w(1-rl)$$

**SRlo:**

Cache misses and invalidations are as in *SRlp*. Writebacks at $P_1$ are initiated by $P_2$ whenever $P_2$ performs a read or write from a state in which $P_1$ currently has a dirty copy of the region, state *01*. Due to symmetry, $E_s$=$E_r$.

$$M = V_{10}[R_1+W_1] = w(1-l)$$
$$Is = V_{10}[R_1+W_1] = w(1-l)$$
$$Es = V_{01}[W_2+R_2] = w[(1-l)w+(1-l)r] = w(1-l)$$
$$Er = V_{10}[W_1+R_1] = w[(1-l)w+(1-l)r] = w(1-l)$$

**SRoo:**

Cache misses are as in *SRlp*. Writebacks are as in *SRlo*. Invalidations are different from *SRop* because a write transition by $P_2$ from state *10* implies both a remote writeback and an optimistic invalidation, and in this case we count the cost as part of a remote writeback. Then, only write transitions by $P_2$ from state *00a* and 00b incur invalidations of $P_1$, and only write transitions by $P_1$ from states *00a* and *00b* cause remote invalidations by $P_1$.

$$M = V_{10}[R_1+W_1] = w(1-l)$$
$$I_s = V_{00a}[W_2]+V_{00b}[W_2] = w(1-l)w+(1-l)r(1-l)w = rw(1-l)$$
$$I_r = V_{00a}[W_1]+V_{00b}[W_1] = w(1-l)w+(1-l)r(1-l)w = rw(1-l)$$
$$E_s = V_{01}[W_2+R_2] = w(1-l)$$
$$E_r = V_{10}[W_1+R_1] = w(1-l)$$

**Dir:**

Cache misses occur on (a) read or writes by $P_1$ from a state in which $P_1$ does not currently have a copy of the data (state *10*), and (b) on writes by $P_1$ from a state in which $P_2$ also has a copy of the data (states *00a* and *00b*).

$$M = V_{10}[R_1+W_1]+V_{00a}[W_1]+V_{00b}[R_1]$$
$$= w[(1-l)r+(1-l)w]+(1-l)r(lw)+(1-l)r(1-l)w$$
$$= w(1-l)+(1-l)wr = w(1-l)(1+r)$$
$$I_r = V_{00a}[W_1]+V_{00b}[W_1] = rw(1-l)$$
$$E_s = V_{10}[W_1+R_1] = w(1-l)$$

# Appendix B

The $N$ processor Markov model, and its solution, is shown in Figure 13. This model is analagous to the two processor system depicted in Figure 2 with the exception that states *01* and *00a* represent $P_1$, and states *10* and *00b* represent the other $N$-1 processors in the system. $Q$ represents the probability that a specific non-local processor performs the next series of accesses, and is equal to (1-$l$)/($N$-1).

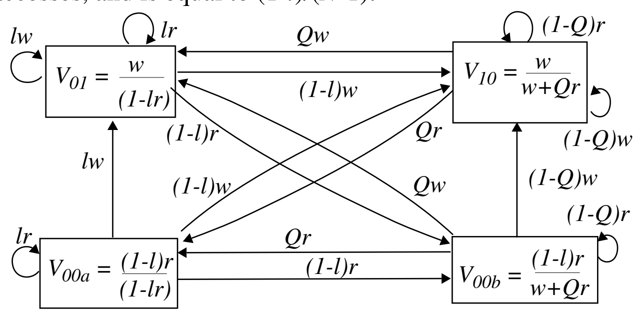



**Figure 13: Symmetric $N$ processor model solution.**