UNIVERSITY OF PISA
DEPARTMENT OF INFORMATION ENGINEERING:
ELECTRONICS, COMPUTERS, TELECOMMUNICATIONS ENGINEERING

Research Doctorate in Electronics, Computers, Telecommunications Engineering
XI CYCLE

EVALUATION OF A COHERENCE PROTOCOL FOR ELIMINATING PASSIVE SHARING IN SHARED-BUS MULTITHREADED MULTIPROCESSORS

Ph.D. Thesis

The Candidate
Eng. Roberto Giorgi

The Advisor
Prof. Eng. Cosimo Antonio Prete

The Advisor
Prof. Eng. Graziano Frosini

The Ph.D. Course Coordinator
Prof. Eng. Umberto Mengali

JANUARY 1999

-

Evaluation of a Coherence Protocol for Eliminating Passive Sharing in Shared-Bus Multithreaded Multiprocessors

Roberto Giorgi
giorgi@acm.org

January 31, 1999

-

Contents

Abstract

Single-chip multiprocessors and multiple-thread architectures are becoming an affordable solution for high-performance general-purpose workstations and servers.

On these machines, the workload is typically constituted of both sequential and parallel applications. Shared-bus shared-memory multithreaded multiprocessor can be used to speed-up the execution of such workload. In this environment, the scheduler takes care of the load balancing by allocating a ready process on the first available processor, thus producing process migration.

Process migration and the persistence of private data into different caches produce an undesired sharing, named passive sharing. The copies due to passive sharing produce useless coherence traffic on the bus and coping with such a problem may represent a challenging design problem for these machines.

Many protocols use smart solutions to limit the overhead to maintain coherence among shared copies. None of these studies treats passive-sharing directly, although some indirect effect is present while dealing with the other kinds of sharing. Affinity scheduling can alleviate this problem, but this technique does not adapt to all load conditions, especially when the effects of migration are massive.

A simple coherence protocol is presented. This protocol eliminates passive sharing using information from the compiler that is normally available in operating system kernels. The performance of this protocol has been evaluated and compared against other solutions proposed in the literature by means of enhanced trace-driven simulation.

The performance of the proposed dolution outperforms the other protocols, especially in the case of a multithreaded processor, thus demonstrating its effectiveness in this kind of hardware platform.

The complexity of the proposed approach has been evaluated in terms of the number of protocol states, additional bus lines and required software support.

The protocol further limits the coherence-maintaining overhead by using information about access patterns to shared data exhibited in parallel applications.

Chapter 1

Introduction

Shared-bus shared-memory multithreaded multiprocessors are a suitable platform to speed up the execution of general-purpose workloads, thus overcoming the intrinsic limitations of uniprocessor systems [Stenstrom97b]. Shared-bus architecture is the straightforward approach to connect several processors having private caches by means of a simple interconnection network. Coherency units have to be added in order to maintain a consistent view of the shared memory for each processor [Flynn95], [Hennessy96], [Hwang93]. Multithreaded processors have the potential for increased performance due to their ability to better tolerate long-latency for memory access [Iannucci94].

When scaling up this architecture, the bus soon becomes the bottleneck of the system. In fact, the bus easily saturates not only because of the traditional bus traffic, present even in cache-based uniprocessors. The bus saturation is also caused by the traffic induced by cache coherency. Part of this traffic can be avoided by using an adequate coherence protocol. Indeed, the performance and the scalability of this architecture mainly depend on the coherency strategy and workload features. In particular, the scalability is heavily dependent on the adaptivity of coherency strategy to the memory access patterns exhibited by running applications [Veenstra92].

Several studies have analyzed the access patterns [Eggers88], [Dubois88], [Dubois91], [Prete97], that depend in great deal on the application itself, highlighting the following main categories of sharing: *active sharing*, *false sharing*, and *passive sharing*. Active sharing involves data actually shared among processes [Gupta92], [Adve91]. False sharing occurs when several different processors access separate data stored in the same memory block. This problem is due to a mismatch in the granularity of sharing of the application and the coherence granularity of the cache [Torrellas90], [Eggers91]. Passive sharing occurs when a private data block is replicated in more than one cache as a consequence of the migration of the owner process [Prete90], [Prete95b], [Prete97].

Passive sharing greatly influences the performance of a multiprocessor running general-purpose

workloads composed of both sequential and parallel applications under a multi-tasking operating system. In previous works, it has been showed that, in this kind of workload, passive sharing has a substantial weight on the overall performance of the system for all coherence protocol schemes [Prete97], [Giorgi97e]. In a write-update coherence scheme the number of write transactions, due to passive sharing, may be as high as 80% of the total write transactions for workloads consisting of only sequential programs that run concurrently. This value does not decrease below 65% when a parallel application is added to the same workload [Prete97]. In these cases, passive sharing may take up 40% and 30% of the bus bandwidth, respectively. This high traffic induced designers to adopt write-invalidate schemes that partially reduce, but do not eliminate, that overhead. After the elimination of this overhead, the reconsideration of write-update schemes is worthwhile [Xia96b].

Process migration induces not only the generation of passive sharing, but also the scarce reuse of cached copies. Mogul and Borg found cache reload overheads of up to 8% of the execution time [Mogul91]. Process migration cannot be avoided, if the system has a scheduler that automatically balances the load among processors [Hwang93]. Even, special scheduling strategies like cache-affinity [Markatos92], [Squillante93], [Torrellas95], [Vaswani91] cannot avoid process migration in all load conditions. In fact, the scheduler could be forced to reduce affinity in order to have a running process on each processor. Indeed, the probability of generating passive copies increases as the time interval between the instant in which the process is suspended from execution and its next rescheduling on a different processor decreases. This interval is statistically small when the number of ready processes is comparable to the number of processors. In this scheduling situation, the cache affinity policy fails.

Different solutions for coherence protocols have been proposed [Stenstrom90], [Tomasevic93], [Tomasevic94]. These solutions are examined here by grouping them in the following categories. Write-Update (WU) protocols distribute the write operation involving a shared copy by using the intrinsic broadcast nature of the shared-bus. Write-Invalidate (WI) protocols maintain coherency by invalidating remote copies upon each write operating on a shared copy. Hybrid (HY) and Hybrid Adaptive (HA) protocols use some kind of switching strategy between a WU and WI behavior. Selective protocols (SE) try to cope with some special problems that affect the performance of the system. None of the above protocols explicitly treats the problem of passive sharing, although some indirect effect is present while dealing with other kinds of sharing. For example, protocols that minimize the overhead of ”migratory sharing” [Cox93], [Stenstrom93], a special case of active sharing, partially reduce also passive sharing effects.

A cache coherence protocol, named PSCR (Passive Shared Copy Removal), can eliminate passive sharing in throughput-oriented multiprocessor. The simple idea of this solution is to locally

invalidate a cached copy belonging to a process private area as soon as the same block is fetched by another processor. This information can be produced easily by the compiler and is normally available in modern kernels and it is used, for example, by memory management mechanism in multi-tasking environment. PSCR can simply use this information without the need of adding extra memory into the cache and the modification of program. The protocol has a reduced complexity since it has only five states, and it needs only an additional line on the bus compared to MOESI protocol scheme [Sweazey86]. I am not aware of other solutions that explicitly eliminate the overhead due to private data accesses. The selective invalidation mechanism allows PSCR to gain the benefits of an update mechanism in shared bus architectures.

To show the effectiveness of PSCR, its performance is evaluated against other protocols either presented in the literature or used in commercial multiprocessors. Process migration shows up in a multitasking environment. For this reason, the present work considers general purpose workloads, which are the usual workloads for the platform under study, instead of purely parallel workloads which are more common in protocol evaluations [Stenstrom93], [Gee93], [Eggers88]. The first of these workloads consists of typical sequential programs like Unix system commands, utilities, and user applications. This scenario simulates the execution of a shell script. In the other two workloads, a parallel application is added to this basic workload to model also a situation in which the user may want to run parallel applications along with the other programs. The selected applications (MP3D and Cholesky) belong to the SPLASH suite [Singh92]. In this scenario, other user generated kind of sharing is also present along with passive sharing.

The performance evaluation has been carried out by using the ”Trace Factory” environment [Giorgi97e], [Prete95]. Trace Factory permits the generation of a combined workload in which the concurrent execution of several applications is simulated, and also includes the most influencing activities of the kernel, namely virtual memory, scheduling and system calls. Moreover, the simulation is then performed using an enhanced trace-driven technique that solves some limitations [Goldschmidt93] of the classical trace driven simulation. Trace-driven simulation offers a good trade-off between speed and accuracy when the performance evaluation’s target is the memory hierarchy and processor interconnection subsystem [Uhlig97].

The protocol sensitivity to architectural parameters such as several processor/bus speed combinations, cache block sizes, set associativity and bus width has also been analyzed. As for the scheduler, conditions, in which the processor affinity is high, have been explored.

Finally, starting from the previous reference analysis, the multiple contexts case is examined and related to the basic analysis.

The protocol performance can be further enhanced if the compiler is able to extract information

about the access patterns to shared items. In this case, the evaluation results showed an improvement when certain blocks belonging to shared areas are treated as if they were private, invalidating them in advance.

The main approaches to achieve cache coherency in bus-based multiprocessor and some solutions that may have effects on passive sharing are reviewed in Chapter 2. In this Chapter an overview of multithreaded architecture solutions and machine is also presented. Chapter 3 presents the new coherence protocol to treat and avoid passive sharing. Chapter 4 presents the multithreaded multiprocessor model used in the simulations. Chapter 5 discusses the methodology used to evaluate the performance of PSCR and six other protocols. In Chapter 6, simulation results are presented and PSCR is compared against others for various case studies. Finally, in Chapter 7, the complexity of PSCR is analyzed and compared with the one of the other protocols.

Chapter 2

Related Work

In this Chapter, the main approaches to achieve cache coherency in bus-based multiprocessors will be recalled and some solutions that may have effects on passive sharing will be highlighted. This list is not intended to be exhaustive. The classification of some solutions is problematic, since the approach may fall in more than one class.

In addition, an overview of multithreaded architectures is presented, considering also some example of academic and commercial machine.

2.1 Coherence Protocols

Across the last fifteen years, many efforts have been made toward a reduction of the overhead introduced by cache coherence. It is now clear that many factors influence the overhead introduced by cache coherence. First, the access patterns [Gupta92], [Brorsson94], [Srbljic97] vary for different data elements in the same application and for different applications. Moreover, data are allocated to memory blocks that may in turn exhibit different aggregations of the original access patterns. Finally, when general-purpose workloads are considered, access patterns generated by migrating processes induce passive sharing[1].

Secondly, the access patterns to data may change during the program execution. This suggests the introduction of some kind of adaptive behavior into coherence schemes. Some processor constructors thus started to introduce some support for adaptivity. The MIPS R4000 [Mips92] allows a per-page dynamic selection of the coherence scheme, the Intel i860XP [Intel91] permits a per-block update policy selection, the DEC Alpha 21064 [Dec93] implements a custom dynamically adaptive protocol.

Thirdly, different choices for the architectural or operating system parameters may weigh dif-

[1] Adve et al. refer to passively shared objects with a different meaning [Adve91]: these are originally shared data that are not actively shared.

ferently on the coherence overhead. For instance, high block sizes may introduce penalties due to false sharing [Torrellas90], [Eggers91], scheduler policies like cache-affinity [Squillante93] may favor the reuse of cached copies and limit passive sharing, variable mapping influences the coherence overhead [Xia96b]. In addition, the architecture itself may condition some characteristic of the access pattern (e.g., block size may change the WRL).

2.1.1 Write-Update and Write-Invalidate Protocols

First solutions coped with coherency by using some kind of clever static strategy: updating or invalidating remote copies upon a write operation involving a shared copy. The Write Invalidate (WI) class consists of those protocols that invalidate remote copies upon a write operation involving a shared copy. In this class early protocols were Write-Once [Goodman83], Synapse [Frank84], Illinois [Papamarcos84], Berkeley [Katz85], RB (Read Broadcast) [Rudolph84], and EIP (Efficient Invalidation Protocol) [Archibald87]. The Write-Update (WU) class consists of those protocols that update remote copies upon a write operation involving a shared copy: Dragon [McCreight84], Firefly [Thacker88], and RST (Reduced State Transition) [Prete91]. A first evaluation of most of these protocols can be found in [Archibald86]. Two new WU protocols have been defined for two special bus-based machines: on-chip multiprocessor [Takahashi96] and bus-based COMA [Lee94].

A first attempt to standardize protocols yielded the MOESI class of protocols, in order to implement them on a common platform [Sweazey86]. MESI is a MOESI class protocol, based on Goodman's Write-Once 4-state protocol [Goodman83] [2] . It is implemented in most of the commercial high-performance microprocessors like AMD K5 and K6, the PowerPC series, the SUN UltraSparc, SGI R10000, Intel Pentium, Pentium Pro, Pentium II and Merced.

2.1.2 Enhanced Write-Update Protocols

Recently, [Takahashi96] introduced a WU protocol, named CRAC (Coherence solution Reducing memory accesses using a Centralized coherency unit) suitable for high-performance on-chip multiprocessors. The protocol is specifically tailored for their architecture and it has five states. They compared the performance against a WI protocol (Berkeley) and two WU protocols (Dragon and Firefly) finding a gain over all the other schemes. They attribute the improvement to both to the centralized coherency unit and the protocol designed for it.

Other new architectures, like the bus-based COMA induced the need to adapt coherence protocols to that architecture where, for instance, replacement implies a block remapping. [Lee94] introduced

[2] Culler [Culler98] reports that MESI was first published by [Papamarcos84], and then named Illinois protocol.

DICE protocol for that machine, which maintains coherency by using a WU scheme. The evaluation shows that a better performance over a WI protocol (Berkeley).

2.1.3 Hybrid Protocols

The performance of a protocol depends on the access patterns exhibited by running programs and, therefore, neither WU nor WI are the best choice for all programs [Eggers88], [Veenstra92], [Veenstra94b]. Eggers introduced two metrics named "*write-run length*" (*WRL*) and "*external rereads*" (*XRR*) to characterize access patterns to shared data [Eggers88], [Eggers89]. The first metric is the number of write operations issued by a given processor to a memory block before another processor accesses that block. (A write-run" is the sequence of write—possibly interleaved by read—references.) The second metric indicates the number of processors that execute read operations on a block, between two consecutive write runs. A natural use of these statistics is to select the better coherence strategy between WI and WU for a given application. A long write-run and a low XRR value suggest that a write-invalidate coherence protocol should be chosen. The cost of the initial misses (caused by invalidation and indicated by the XRR value) is balanced by the large amount of bus traffic saved because all of the subsequent write operations can execute locally. A large number of external rereads and a short write-run indicate that a write-update strategy would be convenient. Of course, the cost of misses and updates plays a decisive role in the strategy selection.

The large variations of WRL and XRR statistics among the programs suggested the introduction of hybrid protocols (HY). Some proposed protocols start with WU strategy but switch to WI as soon as a long write-run is encountered or predicted. Others change their behavior for each program, page, or block and dynamically for the same block. Some protocols use a centralized approach to invalidate the remote copies: the writer processor broadcasts the invalidate command. In other protocols, remote caches decide autonomously to invalidate local copies.

The RWB (Read Write Broadcast) protocol [Rudolph84] is one of the first protocols to exhibit a hybrid nature. After a first write-through on a shared copy, the protocol starts to invalidate.

Karlin et al. proposed the algorithm "Competitive Snooping" upon which a protocol can be implemented. The protocol switches dynamically between WU and WI modality when the cumulative cost of sending updates equals the cost (invalidation threshold) that would be incurred if data had to be read [Karlin86]. They proved that, for any sequence of operations, the overhead of their algorithm is within a constant factor of the minimum required for that sequence.

Eggers and Katz compared a variant of Competitive Snooping, called SR (Snoopy Reading) [Eggers89b] with standard protocols. In SR, on a write to a shared copy, the write operation is

broadcast on the bus and a counter (initialized to 3) in the writer's cache is decremented. When the counter reaches zero, the other copies of the block are invalidated and the counter is reset to three. This protocol has the advantage of using a number of coherence bus-cycles less than twice the number used by the optimal off-line protocol. The authors found good improvement over Firefly for two out of four traces. Berkeley and RB performed better than SR, with the other traces.

Archibald introduced EDWP (Efficient Distributed Write Protocol) [Archibald88] which is similar to Competitive Snooping protocol. In this case the invalidation threshold is fixed at three, but the decision to invalidate a copy is delayed until all counters for that block reach the value of zero (this is detected by means of the shared line). The authors show that this protocol has an improved performance compared to previous HY protocols.

Veenstra and Fowler introduced and evaluated three hybrid protocols that choose the WU or WI modality statically per-page, statically per-block or dynamically per-reference, respectively [Veenstra92]. They found that HY protocols substantially reduce the cost of memory references for most of the program studied. None of the programs receives a significant additional benefit from using a dynamic hybrid protocol compared to the per-block static hybrid protocol, for cache block sizes smaller than 64 bytes.

Gee and Smith proposed a variation of EDWP named "Update-Once" [Gee93]. They evaluated the protocol over a wide range of traces and architectural parameters, and compared the performance against a large set of protocols. They used four WI protocols (Write-Once, Illinois, Berkeley, and Full-MOESI-Invalidate), three WU protocols (Dragon, Firefly, and MOESI-Update), and one HY (EDWP). Update-Once has an invalidation threshold of one, and results show that it yields the highest average performance over the set of protocols.

Another HY protocol is AXP, which is the protocol of Alpha 21064 [Dec93]. The system is configured with two levels of caches that guarantee that if a block is present in L1 cache then it is also present in L2 cache. In this protocol, invalidation is performed on a strict local criterion. Upon an update, if the block is present in both caches, the protocol updates the copy in L2 and invalidates the copy in L1. If the copy is only present in L2, this copy is invalidated. This local invalidation mechanism is also referred to as the "drop rule" [Veenstra94b]. In this case, the invalidation threshold is two. Veenstra and Fowler evaluated AXP, finding that it does not perform better than Illinois [Veenstra94b].

2.1.4 Adaptive Hybrid Protocols

Adaptive Hybrid (AH) protocols dynamically switch between WU and WI policies or are pattern-sensitive, modifying the basic protocol behavior to manage the necessary coherence operations adaptively.

Cox and Fowler introduced an Adaptive protocol with Migratory Sharing Detection (here referred to as AMSD) for bus-based systems [Cox93]. Migratory sharing is characterized by the exclusive use of data for a long time interval. Typically, the control over these data migrates from one process to another [Gupta92], [Stenstrom93]. The protocol identifies migratory-shared data dynamically in order to reduce the cost of moving them. The implementation is an extension of a common MESI protocol. The four basic states are augmented with three new states and an additional bus line is required. The authors evaluated the protocol by considering only the accesses to shared data and by excluding accesses to synchronization variables, private data, and instructions. They observed that ”treating private data as though they were migratory would reduce the cost of process migration”. Stenström et al. also introduced a detection mechanism of migratory sharing for a directory-based write-invalidate protocol [Stenstrom93], with a solution that is very similar to AMSD. The migratory detection mechanism has also been applied to a directory-based competitive-update protocol [Grahn96], [Nilsson94]. This competitive-update protocol is an extension of Competitive Snooping [Karlin86] to directory-based protocols.

Veenstra and Fowler evaluated some variants of the AXP that extend the basic behavior in two directions. One improvement (AXP+) is on the managing of migratory data [Gupta92], and the other (AXPa) on the adaptive behavior. They found that the migratory optimization [Cox93], [Stenstrom93] influences HY protocol performance, more than choosing an optimal threshold for switching from WU to WI modality. The application of migratory optimization on the adaptive behavior (AXPa+) does not introduce improvements to the adaptive behavior extension (AXPa).

Anderson and Karlin introduced two protocols that change their behavior not only on a per-block basis but also dynamically for the same block [Anderson96]. Such protocols are motivated by the large variability of WRL observed for different programs and for different blocks of the same programs. These two protocols are based on the Snoopy Reading protocol. The adaptive behavior is achieved by using an invalidation threshold for each block whose value is adjusted after each write-run, whereas Snoopy Reading protocol uses a constant invalidation threshold. In the RW (Random Walk) protocol, the block threshold is initially zero. The threshold is incremented if the block experiences a WRL not greater than the SR invalidation threshold. It is decremented in the opposite case. The LTS (Last Three Samples) protocol approximates the mean value of the WRL distribution

using the last three WRL samples. They compared the performance of these two protocols against Illinois and Dragon. The performance results were ”in all cases closer to better of WI and WU” [Anderson96].

2.1.5 Selective Protocols

[Matsumoto93] introduced Allread-write snoopy protocol, which can be represented in the framework of conventional MOESI/MESI model. While accessing marked data, on a read-miss, all caches read the block, and on a write operation, the issuing cache performs a write-through and all other caches load the block unless it causes a copy-back transaction. The authors found substantial improvement over a WU protocol (Firefly) and a WI protocol (Illinois), but the evaluation has been carried out only on simple doacross-loop programs. Moreover, additional hardware is required for taking data on allread (allwrite) accesses. The scheme is classifiable in the SE class of protocol, since good results have been shown only for array accesses.

Many protocols have been proposed to solve the problem of false sharing copies [Torrellas90], [Eggers91]. Anderson [Anderson94] introduced SB (Sub-Block) protocol to cope with the problem of false sharing by using a coherence unit smaller than the transfer unit. The transfer block size is large to take advantage of spatial locality. The coherence is maintained on part of the block to avoid false sharing. The base protocol is a variation of Illinois[3] . They compared the protocol against a WI protocol (Illinois) finding some improvement only for particular sizes of the block for each simulated application.

[Kadiyala95] introduced the DSB (Dynamic Sub-Block) and FSB (Fixed Sub-Block) protocols and compared their performance with a WI protocol (Berkeley), finding good improvement over the basic protocol. DSB is an improved version of FSB that dynamically determines the block size on which to maintain coherency.

[Berg95] compared the performance of some snoopy implementations of originally directory-based protocols. The OTF (On-The-Fly)[4] protocol is derived from [Censier78] protocol; the WBWI (Write-Back Write-Invalidate) protocol is derived from PBI (Partial Block Invalidation) protocol [Chen93]. They found a slight advantage in using protocols that treat coherency on sub-blocks.

[Tomasevic92], [Tomasevic96] introduced WIP (Word-Invalidate Protocol) that uses partial invalidation of the coherence block. The protocol invalidates the whole block after that an invalid word threshold has been reached. They evaluated the protocol against a WU protocol (Dragon) and a WI protocol (Berkeley) finding an improvement over Berkeley and sometimes over Dragon.

[3] Read-snarfing is implemented as in [Eggers89], but without locking out the processor from accessing the cache.

[4] See also [Dubois93] for the names of these protocols.

[Xia96b] proposed new hardware and software support to reduce data cache misses in a multiprocessor operating system. Among the techniques, "'data privatization"' is introduced to cope with false sharing. It consists in a reallocation of private variables in different cache blocks. Also, a WU protocol (Firefly) is selectively applied to a subset of variables that are known to exhibit fine-grained sharing. In the other cases, they apply a WI protocol.

SB, DSB, FSB, On-the-Fly, WBWI, and WIP show some positive effect to cope with the problem known as "'false sharing"' [Torrellas90], [Eggers91]. Other techniques exist to cope with this problem, including padding [Eggers91], program transformations [Jeremiassen95], program restructuring [Cheriton91], separate cache block allocation [Torrellas90], and the adoption of some relaxed memory consistency model [Dubois91b].

Other techniques exist to cope with this problem. These include padding [Eggers91], program transformations [Jeremiassen95], separate cache block allocation [Torrellas90], data privatization [Xia96b], and the adoption of some relaxed memory consistency models [Dubois91b].

2.1.6 Lock-based Protocols

Bitar [Bitar86] introduced a protocol that supports synchronization primitives directly. Although they introduced interesting innovations, they did not carry out performance evaluation of this protocol. [Lee90] also described a protocol that supports lock primitives, having 13 states. The bus-based version, called Snoopy CBL (Cache-Based Lock) [Ramachandran96] has been evaluated against a WU protocol (Dragon) a WI protocol (Berkeley) and the previous scheme, called BD (Bitar and Despain, the authors). The results show a good improvement over all these schemes, especially in the case of fine-grained sharing.

2.1.7 Protocols for General-Purpose Workloads

When a multiprocessor is used to speed up a workload that consists of sequential independent applications, the system may a drastical drop in performance [Prete97], [Giorgi97e]. This is due to passive shared copy coherence overhead, i.e. the private data of those programs that become shared because of the process migration. Solutions like cache flushing on context switches generate a substantial increase of cache misses, and inhibit the reuse of cached copies and the adoption of techniques like affinity-scheduling.

Of course the coherence has to be enforced even in this case, since the machine can also be used to concurrently run parallel programs, that exchange data using the shared-memory model. In addition, the system kernel is normally a parallel kernel that runs on each processor and makes use

of shared data structures.

Figures 2.1 and 2.2 show the percentage of bus write operations due to passive and other shared copies in the case of Dragon protocol and two different workloads. The effects of passive sharing are significant in both cases.

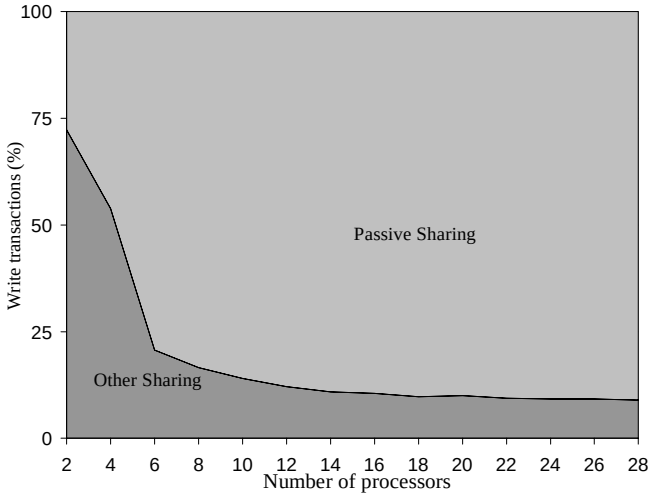

Figure 2.1. Percentage of write transactions due to passive sharing and other sharing in case of the Dragon protocol. The workload consists of a shell script including typical sequential programs (system commands, utilities, and user applications). Each processor has a 256-Kbyte, direct access private cache with 64-byte cache block size. The effects of passive sharing are significant for a number of processors greater than 6.

A first protocol that tried to eliminate the overhead induced by passive sharing was UCR (Useless Copy Removal) [Prete90]. In this case, each cache invalidates locally *unused* copies, as soon as another cache fetches that data block. Cache copies are classified as unused when the copy is not used by the process currently running on the corresponding processor. Since a systematic detection of this condition requires high hardware overhead to store the process identifier for each block, an approximation involving only one extra bit per-block was introduced. The evaluation employing synthetic workloads shows a performance improvement both over RST and Dragon protocol [Prete95].

Skeppstedt and Stenström introduced a variation to Censier and Feautrier protocol [Censier78], named ”private-detection technique” [Skeppstedt95] by inserting a simple heuristic. The protocol classifies blocks as effectively private by observing the transaction sequence involving the block. The results show improvement over the basic protocol. The technique has been evaluated only for CC-NUMA machines.

Prete et al. describe a technique, named USCR (Useless Shared Copy Removal) [Prete95b] similar to UCR, but in this case the selective invalidation is applied to data accesses that the processor signals to be private. The USCR applied to Dragon and Berkeley protocols is then evaluated on parallel applications, showing up substantial improvement of USCR-Dragon but not of USCR-Berkeley over their respective basic version. On these applications, also, USCR-Dragon exhibits a higher performance than USCR-Berkeley does. The reason is that UCR strategy cannot recover the effect of undesirable invalidations on real shared copies. This also was a good indication of the fact that once passive sharing is eliminated WU protocols may continue to be a viable strategy compared to WI ones.

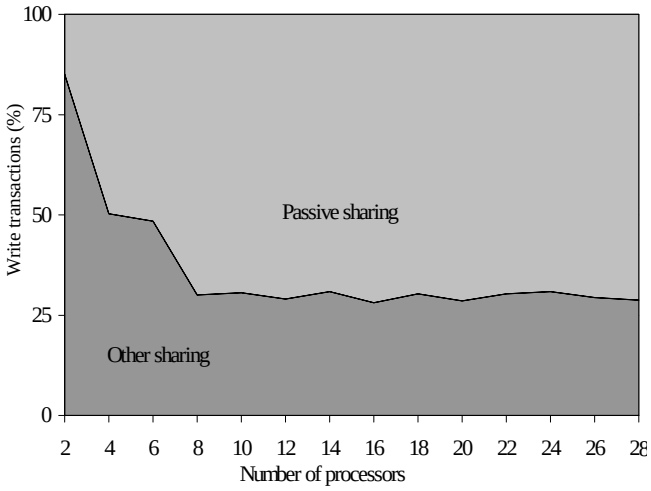

Figure 2.2. Percentage of write transactions due to passive sharing and other sharing in case of the Dragon protocol. The workload consists of a shell script including typical sequential programs (system commands, utilities, and user applications) and the parallel application MP3D. The parallel application and the shared kernel cause write transactions due to true sharing. The number of processes generated by MP3D is equal to 50% of the number of processors. Each processor has a 256-Kbyte, direct access private cache with 64-byte cache block size.

Other mechanisms have been presented to minimize the coherence overhead due to invalidations: DSI [Lebeck95b] eliminates invalidation messages in a directory-based multiprocessor by automatically invalidating its local copy before a conflicting access by another processor. Cheong and Veidenbaum proposed a solution based on the combination of compile-time reference tagging and individual invalidation of potential *stale* copies only when referenced [Cheong88]. Carte et al. used an update time-out mechanism to invalidate replicas of copies that have not been accessed recently upon receipt of an update [Carter95].

2.2 Multithreaded Architectures

Multithreaded architectures have been intensively studied as a way for reducing long latencies in DSM multiprocessors, and may be interesting also for bus-based multiprocessor [Hwang93].

In a single-threaded architecture the computation moves forward one step at a time through a sequence of states, each step corresponding to the execution of one enabled instruction. The state of a single-threaded machine consists of the *memory state* (program memory, data memory, stack) and the *processor state* which consists of *activity specifier* (program counter, stack pointer) and *register context* (a set of register contents). The activity specifier and the register context make up what is also called the *context* of a thread. Today most architectures are single-threaded architectures

According to [Iannucci94], a multithreaded architecture differs from the single-threaded architecture in that there may be several enabled instructions from different threads which all are candidates for execution. Similar to the single-threaded machine, the state of the multithreaded machine consists of the memory state and the processor state; the later, however, consists of a collection of activity specifiers and a collection of register contexts. A thread is a sequentially ordered block of instructions with a grain-size greater than one (to distinguish multithreaded architectures from fine-grained dataflow architectures).

2.2.1 Multithreaded processors

Current microprocessors utilize instruction-level parallelism by a deep processor pipeline and by the superscalar instruction issue technique. A superscalar processor is able to dispatch multiple instructions each clock cycle from a conventional linear instruction stream.

However, the instruction-level parallelism found in a conventional instruction stream is limited. Recent studies showed the limits of processor utilization even of today's superscalar microprocessors [Diep95], [Tullsen95]. To overcome this limitation, the solution is to add more coarse-grained parallelism. The main approaches are the multiprocessor chip and the multithreaded processor. The multiprocessor chip integrates two or more complete processors on a single chip. Therefore every unit of a processor is duplicated and used independently of its copies on the chip.

In contrast, the multithreaded processor stores multiple contexts in different register sets on the chip. The functional units are multiplexed between the threads that are loaded in the register sets. Depending on the specific multithreaded processor design, only a single instruction pipeline is used, or a single dispatch unit issues instructions from instruction buffers simultaneously. Because of the multiple register sets, context switching is very fast. Multithreaded processors tolerate memory latencies by overlapping the long-latency operations of one thread with the execution of other threads

- in contrast to the multiprocessor chip approach. The combination of a multiprocessor chip, which is easier to implement, and the use multithreading is a promising approach.

Research on multithreaded architectures has been motivated by two concerns: tolerating latency and bridging of synchronization waits by rapid context switches. Three different approaches of multithreaded architectures can be distinguished [Silc98]: *cycle-by-cycle interleaving* [Halstead88], [Papadopulos91], [Thistle88], *block interleaving* [Agarwal92], [Sigmund96], and simultaneous multithreading approach [Tullsen95], [Lo97].

2.2.2 Cycle-By-Cycle Interleaving

In the this interleaving model, the processor switches to a different thread after each instruction. An instruction of the same thread is fed in the pipeline after the completion of the execution of the previous instruction. There must be at least as many register sets (loaded threads) available on the processor as the number of pipeline stages. Since cycle-by-cycle interleaving eliminates control and data dependencies between instructions in the pipeline, pipeline conflicts cannot arise and the processor pipeline can be easily built without the necessity of complex forwarding paths. This conducts to a very simple and therefore potentially very fast pipeline - no hardware interlocking is necessary. The latency is tolerated by not scheduling a thread until its reference to remote memory has completed. This model requires a large number of threads and complex hardware to support them. Interleaving the instructions from many threads also limits the processing power accessible to a single thread, thereby degrading the single-thread performance.

In this category follow some machine like the HEP (Heterogeneous Element Processor) [Smith81], the MASA (Multilisp Architecture for Symbolic Applications) [Halstead88], and the Tera MTA (Multi-Threaded Architecture) [Alverson90], [Alverson92].

The HEP system was a MIMD shared-memory multiprocessor system developed by Denelcor inc. (Denver, CO, USA) between 1978 and 1985, and is a pioneering example of a multithreaded machine. Spatial switching occurred between two queues of processes. The main processor pipeline had eight stages, matching the number of processor cycles necessary to fetch a data item from memory in register. Thus, eight threads were in execution concurrently within a single HEP processor. Differently to all other cycle-by-cycle interleaving processor, all threads within a HEP processor shared the same register set.

The MASA was a multithreaded processor architecture for parallel symbolic computation with features intended to facilitate Multilisp program execution. MASA had a tagged architecture, multiple contexts, fast trap handling, and a synchronization bit in every memory word. Its principal

novelty was the use of multiple contexts both to support interleaved execution from separate instruction streams and to speed up procedure calls and trap handling (like register windows).

The Tera MTA is a commercial multithreaded multiprocessor currently produced by Tera Computer Company (Seattle, WA, USA). The machine is a vast multiprocessor with multithreaded processor nodes arranged in 3D mesh of pipelined packet-switch nodes. In the case of the maximum configuration of 4096 nodes, arranged in a 16 X 16 X 16 mesh, there are up to 256 processors, 512 memory units, 256 I/O cache units, 256 I/O processors, and 2816 switching nodes. Each processor is 64-bit custom chip with up to 128 simultaneous threads in execution. It alternates between ready threads, using a deep pipeline. Inter-instruction dependencies are explicitly encoded by the compiler. Each thread has a complete set of registers. Memory units have 4-bit tags on each word, or full/empty and trap bits. The Tera MTA exploits parallelism at all levels, from fine-grained instruction-level parallelism within a single processor to parallel programming across processors, to multiprogramming among several applications simultaneously. Consequently, processor scheduling occurs at many levels, and managing these levels poses unique and challenging scheduling concerns.

2.2.3 Block Interleaving

In the block-interleaving approach, like in the MIT Sparcle processor [Agarwal93], a single thread executes until it reaches a long-latency operation, such as a remote cache miss or a failed synchronization, and then it switches to another context. The Rhamma processor [Gruenewald96], switches contexts whenever a load, store or synchronization operation is discovered.

In this model, a thread is executed for many cycles before context switching. Context switches are used only to hide long memory latencies since small pipeline delays are hidden by proper ordering of instructions performed by the optimizing compiler. Since multithreading is not used to hide pipeline delays, fewer total threads are needed and a single thread can execute at full speed until the next context switch. This may also simplify the hardware. There are three variations of this model: *switch-on-load*, *switch-on-use*, *explicit-switch*.

The *switch-on-load* version switches only on instructions that load data from shared memory, whilst storing data in shared memory does not cause context switching (since local memory loads and other instructions all complete quickly and can be scheduled by the compiler). However, context switches sometimes occur sooner than needed: if a compiler ordered instructions so that a load from shared memory was issued several cycles before the value was used, the context switch should not have to occur until the actual use of the value. This strategy is implemented in the *switch-on-use* model [Mankovic87]. In this case, a valid bit is added to each register. The bit is cleared when the

loading from shared memory to the corresponding register is issued and set when the result returns from the network. A thread context switches if it needs a value from a register whose valid bit is still cleared. A benefit of this model is that several load instructions can be grouped together thus prefetching several operands of an instruction. Instead of using valid bits, an explicit context switch instruction can be added between the group of loads and their subsequent uses. This model, which is called *explicit-switch* [Culler91], is simpler to implement and requires only one additional instruction.

In [Boothe93], it is shown that for most applications just two or three threads per processor is sufficient, once that the cache interaction is taken into account. There are three variations of the cache-extended block interleaving model. The *switch-on-miss* model [Agarwal91] context switches if a load from shared memory instructions misses in the cache. Such a context switch is not detected immediately, however, so that a number of subsequent instructions have already entered the CPU pipeline and thus wasted CPU time. The *switch-on-use-miss* model [Gupta91] context switches when an instruction tries to use the (still missing) value from a shared load that missed in the cache. The *conditional-switch* model provides the benefits of grouping (of the explicit-switch model) and caching. In this model, the explicit switch instruction is ignored if all load instructions (in the preceding group) hit the cache; otherwise, the context switch is performed. Some representative machines in this category are listed below.

The CHoPP [Mankovic87] was a shared memory MIMD with up to 16 powerful computing nodes. High sequential performance is due to issuing multiple instructions on each clock cycle, zero-delay branch instructions, and fast execution of individual instructions. Each node can support up to 64 threads.

The MIT Alewife multiprocessor [Agarwal95], [Agarwal93], [Agarwal90], is based on the multithreaded Sparcle processor. The Sparcle processor is derived from a Sparc RISC processor. The eight overlapping register windows of a Sparc processor are organized in four independent non-overlapping thread contexts, each using two windows (one as register set, the other as a context for trap and message handlers). Thread switching is triggered by external hardware when a remote memory access is detected. Emptying the pipeline from instructions of the thread that caused the context switch and organizational software overhead sum up to a context switching penalty of 14 processor cycles. The Alewife multiprocessor has been operational since May 1994. It uses a low-dimensional direct interconnection network. Despite its distributed-memory architecture, Alewife allows efficient shared-memory programming through a multilayered approach to locality management. Communication latency and network bandwidth requirements are reduced by a directory-based cache-coherence scheme referred to as LimitLESS directories. Latencies still occur although

communication locality is enhanced by runtime and compile-time partitioning and placement of data and processes.

The MSparc has an approach similar to the Sparcle processor is taken at the University of Oldenburg (Germany) with the MSparc processor [Miksch96]. MSparc supports up to four contexts on chip and is compatible to standard Sparc processors. Switching is supported by hardware and can be achieved within one processor cycle. The multithreading policy is block interleaving with the switch-on-cache-miss policy as in the Sparcle processor.

The MIT Jellybean Machine (named J-Machine) [Noakes93]. Its name is due to the fact that it was built entirely of a large number of of Jellybean components. The initial version used an 8 X 8 X 16 cube network, with possibilities of expanding to 64K nodes. The Jellybeans are message driven processor (MDP) chips, each of which has a 36-bit processor, a 4K word memory, and a router with communication ports for 6 directions. External memory of up to 1 M words can be added per processor. The MDP creates a task for each arriving message. In the prototype, each MDP chip has 4 external memory chips that provide 256K memory words. It is possible to implement a shared memory model using message passing, in which a message provides a fetch address and an automatic task sends a reply with the desired data.

The multithreaded Rhamma processor [Gruenewald96] from the University of Karlsruhe (Germany) uses a fast context switch to bridge latencies caused by memory accesses or by synchronization operations. Load/store, synchronization and execution operations of different threads are executed simultaneously by specialized functional units within the processor. The units are coupled by FIFO buffers and access different register sets. Each unit stops the execution of a thread when it recognizes an instruction intended for another unit. To perform a context switch the unit passes the thread tag to the FIFO buffer of the unit that is appropriate for the execution of the instruction. Then the unit resumes processing with another thread of its own FIFO buffer. The Rhamma processor is most similar to the Sparcle. However, the execution unit of the Rhamma processor switches the context whenever it comes across a load, store or synchronization instruction, and the load/store unit switches whenever it meets an execution or synchronization instruction (switch-on-load policy). In contrast to Sparcle, the context switch is in an early stage of the pipeline, thus decreasing context switching time. On the other hand, the overall performance of the Rhamma processor suffers from the higher rate of context switches unless the context switch time is very small. Specific implementation techniques reduce switching costs to zero or at most one processor cycle. These techniques use a context switch buffer which is a table containing the addresses of instructions that already yielded a context switch.

2.2.4 Simultaneous Multithreading

Simultaneous multithreading approach [Tullsen95], [Lo97], or multithreaded superscalar approach [Sigmund96] combines a wide issue superscalar instruction dispatch with the multiple context approach by providing several register sets on the multiprocessor and issuing instructions from several instruction queues simultaneously. Therefore, the issue slots of a wide issue processor can be filled by operations of several threads. Latencies occurring in the execution of single threads are bridged by issuing operations of the remaining threads loaded on the processor. In principle, the full issue bandwidth can be utilized.

The multithreaded processor of the Media Research Laboratory of Matsushita Electric Ind. (Japan) [Hirata92] is the first approach to simultaneous multithreading. Instructions of different threads are issued simultaneously to multiple functional units. Simulation results on a parallel ray-tracing application showed that using 8 threads a speed-up of 3.22 in case of one load/store unit and of 5.79 in case of two load/store units can be achieved over a conventional RISC processor. However, caches or TLBs are not simulated, nor is a branch prediction mechanism.

The simultaneous multithreading approach from the University of Washington (Seattle, WA, USA) [Eggers97] surveys enhancements of the Alpha 21164 processor and of a hypothetical out-of-order issue superscalar microprocessor that resembles R10000 and PA-8000. Simulations were conducted to evaluate processor configurations of an up to 8-threaded and 8-issue superscalar. This maximum configuration showed a throughput of 6.64 instructions per cycle due to multithreading using the SPEC92 benchmark suite and assuming a processor with 32 functional units (among them multiple load/store units) [Tullsen95]. A second approach evaluated more realistic processor configurations and reached a throughput of 5.4 instructions per cycle for the 8-threaded and 8-issue superscalar case.

The multithreaded superscalar processor approach from the University of Karlsruhe [Sigmund96] is based on a simplified PowerPC 604 processor. The processor uses various kinds of modern microarchitecture techniques like separate code and data caches, branch target address cache, static branch prediction, in-order dispatch, independent execution units with reservation stations, rename registers, out-of-order execution, and in-order completion. Using the same number of threads, the multiprocessor chip reaches a higher throughput than the multithreaded superscalar approach. However, if the chip costs are taken into consideration, a 4-threaded 4-issue superscalar processor outperforms a multiprocessor chip built from single-threaded processors by a factor of 1.8 in performance/cost relation.

Chapter 3

PSCR Protocol

None of the protocols described in the previous Chapter (except for UCR [Prete90] and USCR [Prete95b]) explicitly addresses the problem of passive sharing, although some indirect effect is present while dealing with other kinds of sharing.

The reason why those protocols are not effective is briefly explained below. WU protocols have the worst behavior, since they update passive copies until dropped because of replacement, generating a huge amount of unnecessary traffic. WI protocols typically invalidate passive copies on the first write. Thus, they avoid part of the useless traffic, but this indiscriminate invalidation does not allow us to take full advantage from the possible remote use of actively shared copies. Therefore, the more these protocols are effective in treating passive sharing, the more they lose by invalidating actively shared copies. HY and AH protocols have a certain delay in recognizing passive copies, and they limit the possible benefits of avoiding some extra update operation. Selective and Adaptive Hybrid protocols that are able to detect migratory sharing, may produce some positive effects on passive sharing.

Moreover, PSCR seems to be effective in eliminating passive sharing not only in the case of single-threaded multiprocessor architectures, but also as the number of contexts is increased. In this Chapter, the idea behind the protocol is described, while its effectiveness in the multithreaded case will be evaluated in Chapter 6.

3.1 Description of the Idea

PSCR solution, to treat passive sharing, is simple and straightforward. The selective invalidation mechanism allows PSCR to eliminate passive sharing and to gain the benefits of an update mechanism in bus-based architectures. The idea consists in invalidating the copies belonging to private data areas of a process as soon as they are fetched by another processor. Such blocks are referred to as *P-blocks*, whilst *S-blocks* are blocks belonging to code or shared data area.

PSCR ensures that a P-block is never involved in a write transaction. When a cache miss involves a P-block, the only other copy, possibly left by the migrated process in a remote cache, will be immediately invalidated. In this way, private data blocks are gradually forced to ”follow” the owner process in its migration, in order to cause no further coherence-related activity.

3.2 Basic Hardware and Software Support

The proposed approach is both hardware and software based. Private data are supposed to be allocated into separate memory page at compile time. At loading time, the memory management unit uses an extra bit (P-bit) for each page descriptor to indicate if the current page belongs to private data. Compilers and operating system kernels of multiprogrammed environments normally perform this activity in order to manage virtual memory. Thus, no extra software support is required in respect to what is normally present. Moreover, no extra information is necessary in program code.

In Section 6.7, an advanced strategy at software level that can increase the protocol performance is discussed. In Section 7.3, the compiler effort in detecting private data is also discussed.

The hardware implementation is quite simple: the processor uses a dedicated line of the processor-cache bus to signal, on every memory reference. The shared bus should provide the following bus transactions:

- *Read-Block transaction*: the cache loads a copy of a memory block that it does not hold yet. Unless specified, if the copy is furnished by a cache Read-Block stands for *Cache Read-Block transaction*, whilst if the copy is furnished by the memory, *Memory Read-Block transaction.*
- *Write transaction*: the cache broadcasts the contents of a single location on the shared bus.
- *Update-Block transaction*: the cache writes back an entire copy that has to be destroyed.

In the case of miss condition, the cache broadcasts the P-block/S-block information on the common bus by means of a line (L_1) during the read-block transaction. If the transaction involves a P-block and a remote cache holds a copy of that block, the copy is immediately invalidated.

A second line (L_2, handled by the ”listening” caches) is required for a couple of (mutually excluding) purposes: during a read-block or a write transaction involving a S-block (L_1 = OFF),

- to indicate that a copy is resident in at least one remote cache, so that the state of the loaded or newly-written copy must be set to one of the shared states (see below). Other protocols use a line (named shared-line) with the same issue. during a read-block transaction involving
- a P-block (L_1 = ON), to indicate that a dirty copy is resident in a remote cache, so that the state of the new loaded block must be set to Private-Dirty (see below).

A cached copy may be in four valid states:

- *Private-Clean* (PC): only one memory block copy exists and it is consistent with main memory.
- *Private-Dirty* (PD): only one memory block copy exists, but it has been modified and it is no longer consistent with main memory. In case of replacement, the cache must first update main memory.
- *Shared-Clean* (SC): several memory block copies may exist; they are identical but may be inconsistent with main memory.
- *Shared-Dirty* (SD): several memory block copies may exist (one SD and the others SC); they are identical but are not consistent with main memory. The cache with the SD copy must update main memory, if the SD copy has to be destroyed for replacement.

Furthermore, there is the Invalid (I) state, to which a P-block is set after a local invalidation (see below).

3.3 Activities Due To local Processor Operations

The description of a specific coherence protocol must generally consider a couple of independent aspects: type of access (read/write) requested by the local processor, and cache condition (hit/miss). In the case of the PSCR protocol, a further distinction needs to be made between operations on P-blocks and S-blocks. Therefore, eight different cases should be considered. Of course, for two of them (read hit on either kind of block) no coherence action is needed. As for the remaining six cases, a detailed description, with the help of the state diagrams (Figure 3.1) is provided below.

- *Write hit on P-block*: the cached copy is updated. If the copy is PC, its state is changed to PD. If it is already PD, no state transition is necessary.
- *Write hit on S-block*: the cached copy is updated. If the copy is PC, its state is changed to PD. If it is already PD, no state transition is necessary. If the copy is either in SC or in SD state, a bus write-transaction is used to update both the main memory and the copies that may exist in remote caches. Because of this transaction, if L_2 line is not active (the block is no longer shared), the copy state changes to the corresponding private state (SC to PC, SD to PD).
- *Read miss on P-block*: first, a cache block may have to be chosen for replacement. If the victim block is in either PD or SD state, an update-block transaction is used to write back

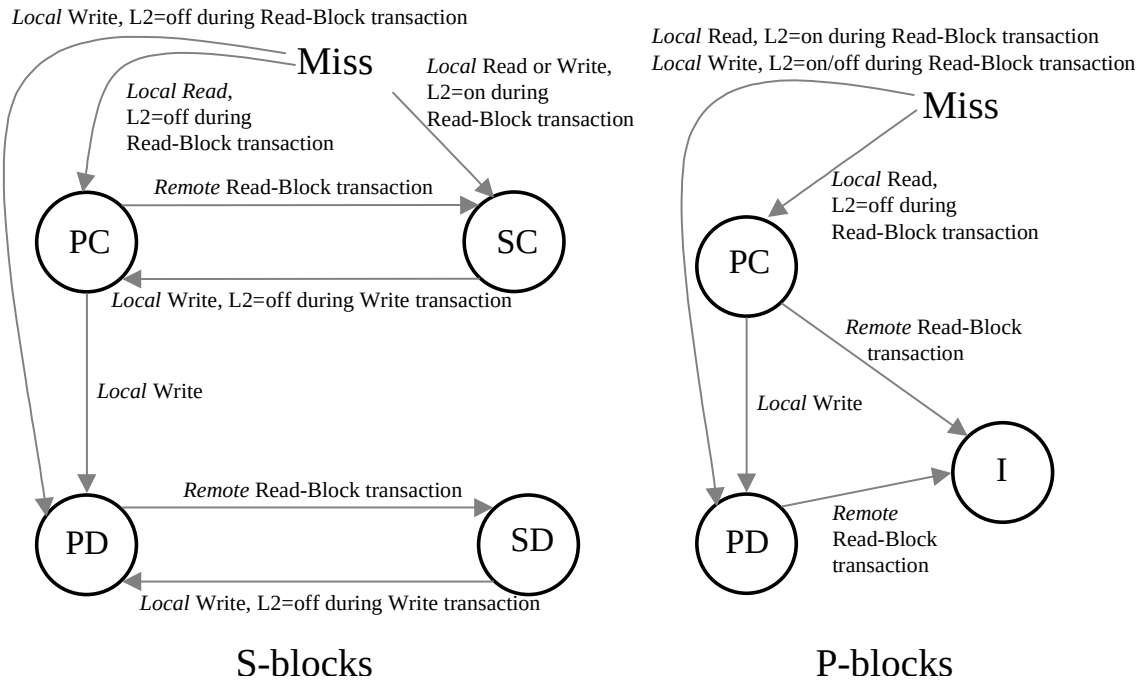

Figure 3.1. State transition diagrams for PSCR protocol.

the modified block into the main memory. Afterwards, L_1 line is activated and a read-block transaction is used to load the missing block. The block is loaded in PD state if L_2 line is activated by a remote cache during this transaction; otherwise, it is loaded in PC state. Finally, the cache supplies the processor with the contents of the involved location.

- *Read miss on S-block*: first, a replacement phase may be necessary as in the previous case. During the read-block transaction needed to load the missing block, L_1 line is not activated. The block is loaded in SC state if L_2 line is activated by a remote cache during this transaction; otherwise, it is loaded in PC state. Finally, the cache supplies the processor with the contents of the involved location.

- *Write miss on P-block*: first, a replacement phase may be necessary, as above. During read-block transaction, L_1 line is activated. The block is loaded in PD state and the pertinent location is updated.

- *Write miss on S-block*: first, a replacement phase may be necessary as above. During read-block transaction, L_1 line is not activated. The block is loaded in SC state if the L_2 line is activated by a remote cache during this transaction; otherwise, it is loaded in PD state. Finally, the pertinent location is updated and, if the copy is SC, a write transaction is performed.

3.4 Snooping activities

During each read-block or write transaction, each listening cache checks whether it holds a copy of the memory block involved. If so, it operates as follow.

In the case of a *read-block transaction*, two different cases can happen:

- if the transaction involves an S-block (line L_1 off), the cache activates L_2 line. If the state is private (PC or PD), then it is changed to shared (PC to SC, PD to SD). If the initial state was PC, PD, or SD, the cache disables the main memory and supplies the data.
- the transaction involves a P-block (line L_1 on), the copy state is set to Invalid. If the cache has a PD copy of the block itself, L_2 line is also activated. Finally, the cache disables the main memory and supplies the data.

In the case of a *write transaction* on an S-block (either if the copy state is SC or SD), the cache updates the copy and no state transition is performed.

Chapter 4

Multithreaded Multiprocessor Model

As seen in Chapter 2, many solutions have been proposed in order to exploit the available chip area for building a multithreaded architecture.

Since this work aims to evaluate the effectiveness of a coherence solution, that is its impact on the memory hierarchy, the multithreaded multiprocessor model is presented in a generic basic version, thus avoiding to model many sophisticate internal details.

Therefore, among the three classes described in Section 2.2 (*cycle-by-cycle interleaving*, *block-interleaving*, *simultaneous multithreading*), the focus of this work is on block-interleaving, since cycle-by-cycle interleaving requires a very high number of threads and a more complicated hardware [Tullsen95], and simultaneous multithreading requires a detailed model of the pipeline (which is not in the scope of this work).

As seen, block-interleaving [Agarwal93] can be used to switch among available hardware contexts, once a long-latency operation has been reached.

The number of threads supported by each processor, the cost of context switch between threads, and the context switch strategy are the parameters which the simulator has to be supplied with. In particular, three different context-switch policies are considered: *switch on miss* [Agarwal91], *switch on read miss*, and *switch on block of instructions* [Muller96]. These techniques involve different cache behaviors with respect in particular to the resulting miss rate and the overhead induced by the coherence protocol [Hwang93].

From a performance point of view, to achieve both lower execution time and higher scalability the designer of a multithreaded multiprocessor may have to cope with some constraint like scarce parallelism or having to change the priority of some critical process.

Once a switch is decided, one of the available contexts is activated and several options have been presented in literature: *thread priorization* [Fiske95], *MRU thread* [Chen96], but generally a *round*

robin is used.

In this multithreaded architecture model, each thread is assigned its own T_{slice}, and the system assumes the presence of $N_{cpu} \times N_{thr}$ virtual processors. Two levels of scheduling need to be considered: i) an external scheduling, concerning the dynamic allocation of threads on the available processors in the system, and ii) and internal scheduling, concerning the context switching between threads on a single processor. The internal level of scheduling is completely handled by the simulator and it is a simple round-robin policy.

Once that the context switch has been completed the new thread can continue its execution until, in its turn, it encounters a long-latency operation. If the previous thread was experiencing a long-latency operation delay or it was waiting for bus-access, the new thread can completely overlap its execution with this wasted time of that thread.

Multithreading helps on latency hiding, but also bus traffic reduction plays an important role. A protocol aware of the application sharing pattern can reduce the number of bus operations [Gupta92].

A point of force where to concentrate efforts is the reduction of bus traffic: multithreading helps on latency hiding but also the number of bus operations should be reduced as much as possible. Some bus operations can't be avoided, but much of the coherence overhead can be reduced by using a protocol adequate for the sharing pattern exhibited by the running applications [Gupta92].

Chapter 5

Methodology

The methodology used in the following analysis is based on trace-driven simulation [Eggers90], [Stunkel91], [Hollyday92], [Prete95], [Uhlig97]. To ensure accuracy, the kernel activities that most affect the performance are simulated. Memory references include both user and kernel references, and they are produced ”on-demand.’

Three kernel activities are simulated: system calls, process scheduling, and virtual-to-physical address translation. Reference bursts, due to system calls, affect performance interrupting the locality of the memory reference stream of the running process. Virtual-to-physical address translation may change program localities that influence the number of ”intrinsic-interference” (or ”conflict”) missescaused by interferences among several accesses in the same cache set. Process scheduling influences the process migration and, as consequence, passive sharing.

The Trace Factory environment [Giorgi97e] has been used to achieve the flexibility needed to perform complex evaluations. The approach used in this environment is to produce a *source* trace (a sequence of memory references, system-call positions and synchronization events in case of parallel programs) by means a tracing tool (Tangolite [Goldschmidt93b], in the evaluation carried out in this paper). Then, Trace Factory models the execution of complex workload by combining multiple source traces and simulating system calls, process scheduling and virtual-to-physical translation. Finally, Trace Factory produces the references (*target* trace) furnished as input to a memory-hierarchy simulator [Prete95].

Trace Factory generates references according to the on-demand policy: it produces a new reference when the simulator requests one, so that the timing behavior imposed by the memory subsystem conditions the reference production [Giorgi96]. It simulates system calls by including synthetically generated memory reference bursts. Process management is modeled by simulating a scheduler that dynamically assigns a ready process to a processor. The process scheduling is driven

by time-slice for uniprocess application, whilst it is driven by time-slice and synchronization events for parallel applications. Virtual-to-physical address translation is modeled by mapping sequential virtual pages into non-sequential physical pages.

In the simulations below, kernel references and bursts are modeled gathering statistics from a set of traces distributed by Carnegie Mellon University and obtained on an Encore Multimax (shared-bus multiprocessor) machine [Vashaw93]. As for the bursts, statistics regarding their length and inter-burst distance have been collected. An evaluation of this methodology has been carried out in [Prete95], [Giorgi97e]. As for the scheduler, input parameters are the time slice in terms of number of references, and the process-scheduling policy (”cache affinity” or ”random”).

In the following, the problems related to tracing issues will be treated in detail, presenting the main characteristics of Trace Factory.

5.1 Performance Evaluation Methodologies and Tools

Many multiprocessor performance evaluation methodologies and tools have been developed by academic, research and commercial entities during recent years. Two main categories of tools can be distinguished: in one set, the tools used to perform tuning of applications executed on a specified computer model, and in a second as well important set, the tools for evaluating different architectural solutions while varying software features. In the former category, important initiatives for High Performance Computing like the ones from NASA, DARPA and NSF have highlighted the need of such tools and a comprehensive state of the art list can be found in the two companion issues of *IEEE Computer* [Pancake95] and *IEEE Parallel and Distributed Technology* [Pancake95b]. Other recent tools range from simple software extensions of the processor or operating system monitoring capabilities (PatchWrx [Perl96]), to more sophisticated and articulated environments (AIMS [Yan96]). As for commercial companies DEC’s ATOM, Intel’s ParAide, CRI’s MPP Apprentice, Convex’s CXpa have been developed, but also some academic project like Paradyn [Miller95] have become a multi-platform performance tuning software.

The focus is mainly on tools that put into the hand of computer architects the possibility of tuning the memory hierarchy and system parameters [Prete95] by selecting an arbitrary and ”ad hoc” workload to stress the machine. In this direction a number of different strategies have been used in the literature: analytical/stochastic models, trace-driven simulation, complete system simulation, just to mention the most common solutions. A classification of the techniques can be issued on the basis of different metrics, such as accuracy of evaluation, cost of implementation, speed, flexibility of the method with respect to a wide range of architectures. In the present Section few different kinds of approach to this issue are mentioned, without any claim to be exhaustive.

The analytical/stochastic model [Vernon88] appears to be the most flexible and economic solution, but the low degree of accuracy which it provides may be unacceptable in the case of cache performance evaluation, since the model does not typically include all aspects which characterize cache and program behavior. This kind of analysis could be of some use in obtaining a quick estimation of system performance, before performing the actual evaluation by means of a more reliable technique.

The methodology based on complete system simulation is the most flexible and accurate, since it potentially allows a detailed analysis of all hardware and software aspects involved in a particular architecture, including a full operating system. The major problem in such methodology is that an extremely complex model is generally required to simulate the execution of sophisticated software such as operating systems or multiprogrammed workloads. As a consequence, a large dilation factor is usually to be expected, particularly when a high detail of simulation has been selected (SIMICS [Magnusson95], SimOS [Rosenblum95]). In particular, when dealing with a multiprocessor system, the slowdown scales linearly with the number of CPUs being simulated. In the case of the SimOS approach, the level of detail of the simulation can be dynamically controlled by the user in order to minimize the total simulation time. With the deepest level of detail, a slowdown factor in the thousands occurs when simulating systems with 16-32 processors [Rosenblum95]. On the other hand, MINT [Veenstra94] provides a set of simulated processors that run standard Unix executable files compiled for a MIPS R3000-based multiprocessor. Spinlocks, semaphores, barriers, shared memory and most Unix system calls are supported. Processors generate multiple streams of memory reference events that drive a user-provided memory system simulator.

When the target of the performance evaluation is the memory hierarchy and processor interconnection subsystem, a good trade-off between speed and accuracy is represented by trace-driven simulation [Prete95], [Eggers89b]. This method is based on the production of a trace (sequence of memory references generated by the running program) and on the utilization of the trace as input for the simulator of the memory hierarchy. Two critical issues concerning accuracy are: i) traces must include both user and kernel references, and ii) a minimal amount of time distortion must be induced either by the tracing mechanism (during the recording phase) or by the simulator (in the utilization phase).

5.2 Trace Generation Issues

Tracing techniques include hardware and software solutions; a detailed schematization appears quite problematic, since each tool presents at least one feature which makes it unique and not suitable to be assimilated to other solutions to be classified into the same group.

Hardware monitoring is the solution which can potentially guarantee best results in terms of accuracy. Vashaw and Wilson [Vashaw93] used this technique to collect the traces by means of a couple of identical machines (two Encore Multimax 320s were employed). The procedure is to record at full speed the references generated by the traced machine into the memory of the tracing machine until the trace memory is exhausted; then, the tracing machine starts storing the traces. Timestamps are inserted at synchronization points to allow the correct replay of the collected traces. Accuracy, absence of time distortion and of intrusiveness are the main advantages of this method. The most critical drawback of this approach comes from the fact that modern trends in technology for processors encourage the adoption of on-chip caches, so that a large number of memory references are handled internally and can no longer be captured by the hardware tracing mechanism [Stunkel91]. Other limiting factors are the high cost of implementation and the lack of completeness (fragmentation) of the trace gathered, due to the limited size of storage buffers. Finally, traces obtained from an actual multiprocessor machine by means of hardware techniques cannot be employed for an exhaustive performance analysis of the system, because it is not possible to produce traces with a variable number of processors.

Software tracing methods include *program instrumentation*, *single-step execution*, and *microcode modification*. A major problem, which at various levels affects all these methods, is *time dilation*, due to the fact that the software tracing mechanisms generate a heavy overhead, which causes major changes in the relative timing of asynchronous events, resulting in a lower accuracy compared to hardware approaches.

In program instrumentation, a set of instructions are added to create the portion of the trace relative to each *basic block* (sequence of machine-level instructions not containing branches) throughout the program. The instrumentation phase may be activated at either source or executable level. The latter is simpler to handle for the user, but it is quite difficult to implement and could not allow to instrument all programs; on the other hand, instrumentation at assembly level is quite easy, but the user must have access to the entire source code. Also, the accuracy of the model is limited by the lack of completeness in the trace, since it is quite difficult to capture references of kernel routines.

MPTRACE, TRAPEDS and TangoLite are some examples of tracing tools based on program instrumentation. MPTRACE [Eggers90], a tool for collecting traces of multithreaded parallel programs, was developed by Eggers et al. for Sequent i-386 shared memory multiprocessor. It automatically modifies the assembly language version of the application, inserting code to collect traces. The TRAPEDS [Stunkel92] tracing system was originally developed by Stunkel and Fuchs for the Intel iPSC/2 hypercube multicomputer. This version traces both user and kernel code, and performs simulation on-the-fly to avoid large storage costs. A later extension was implemented on a 8-node

bus-based multiprocessor (Encore Multimax 510). To guarantee the accurate recreation of the interactions between processors, TRAPEDS uses a timer-based approach. TangoLite [Goldschmidt93] is a software instrumentation system for the MIPS instruction architecture, developed by Goldschmidt et al. It supports the execution-driven ("on-the-fly") simulation of multiprocessor workload, and it can also generate multiprocessor traces. The instrumentation is performed mostly at the assembly level. The processes are represented as light-weight threads; the scheduling policy guarantees that events generated by different processors are simulated in chronological order.

The problem of completeness arises in single-step execution. This technique can be adopted with microprocessors which allow the execution of a program to be interrupted after each instruction. Nevertheless, since kernel routines typically disable interruptions, no possibility usually exists to capture references generated by the execution of kernel routines. In the trace generator implemented by Eggers and Katz for the Sequent architecture [Eggers88], the tracing mechanism uses trace-trap facilities to halt at each instruction and dump trace information, both for instructions and their operands.

The tracing technique based on microcode modification (ATUM) uses processor microcode to record references in a reserved part of main memory as a side effect of normal execution [Sites88]. Compared with other techniques, this one leads to fewer distortions and a very fast recording (only 10x slowdown); all the system activities can be observed, with no additional hardware being required. The disadvantages include poor flexibility, since microcode modification requires access to on-chip ROM. Furthermore, the trace length is limited to the amount of the memory reserved for the trace storage.

In any case, when the goal is to compare different architecture solutions, it becomes important to analyze the system behavior under predefined and controlled workloads. Two key points of this approach are: i) traces must represent actual workloads for the target machine, and ii) the designer must have the possibility to produce proper traces to investigate the behavior of the system when exposed to particular (possibly critical) workload conditions. This kind of flexibility can be guaranteed only by software techniques, and this is the main reason why this direction has been taken.

5.3 Overall Description of Trace Factory Environment

Trace Factory is an operating environment to create traces representing a specific user workload executed on a specific multiprocessor configuration with a particular kernel behavior. Figure 5.1 show an overview of the processing flow of this environment.

Trace Factory allows the utilization of a set of *source* traces including only user references to

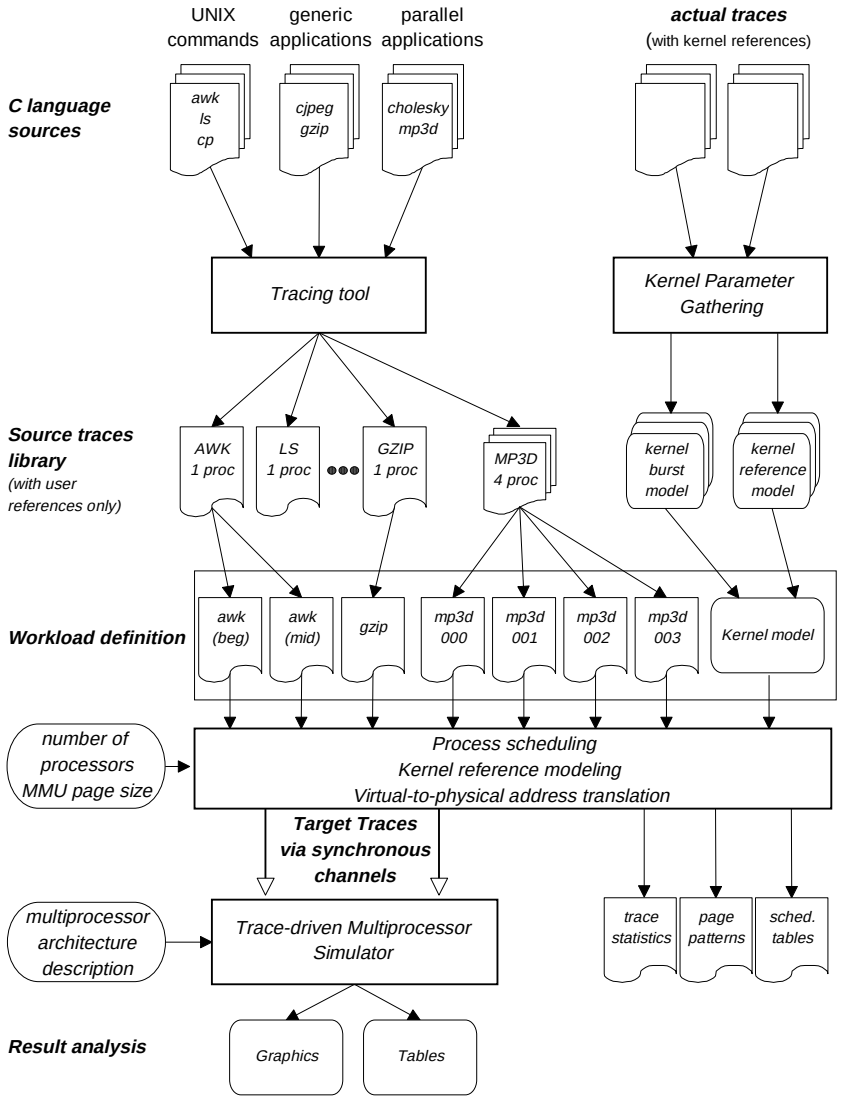

Figure 5.1. The global scheme to produce and to use a target trace in a simulation.

produce complete multiprocessor *target* traces. Source traces can be obtained by a tool [Stunkel91] based on the same microprocessor used in the target system. (e.g. TangoLite [Goldschmidt93] may be used to study a MIPS-based workstation.) Target traces are generated by considering the source traces, the target machine configuration (e.g. the number of processors) and the following three kernel aspects: i) *kernel memory references*, i.e., the reference bursts due to each system call and kernel management routine; ii) *process scheduling*, i.e., the dynamic assignment of a ready process to an available processor; and iii) *virtual-to-physical address translation*, i.e., the mapping of virtual addresses, produced by a running process, to physical memory addresses. The reference sequences can be simply stored into target trace files or supplied to the multiprocessor simulator via

synchronous channels; in the latter case, the target trace generation is performed on the basis of the *on-demand* policy: a new reference is produced when requested by the simulator, so that the trace generated is conditioned by the temporal behavior imposed by the memory subsystem.

5.3.1 Generation of Kernel References

Kernel reference bursts affect performance because they interrupt the locality of the memory reference stream of the running process causing additional cache misses. In this hybrid approach, the kernel reference stream is obtained by means of a stochastic model of addresses and burst positions.

Kernel bursts are obtained by inserting sequences of kernel references within the user reference stream. These sequences are generated by means of two statistics: *length* of each burst and *distance* between the starting point of two subsequent bursts. Each kernel reference is specified by: *area referenced* (code/data), *address* within the selected area and *kind of access* (read/write). The probability of code/data access and of data read/write access are input parameters for the tool, together with additional locality information.

To obtain these statistics, the proposed method is to start from a trace which includes kernel references, and extract the parameters that describe the model of kernel behavior. In particular, the probability of code/data access and of data read/write access can be directly evaluated by counting the relative occurrences of events. Concerning the locality of memory references, the following parameters can be separately evaluated for code and data areas: i) the maximum distance between two subsequent references; ii) the maximum amplitude of the distribution of distances between two subsequent references; iii) the percentage of backward references over the total number of non-sequential accesses. With the same procedure described in [Prete95], these parameters are used to set up the shape of an empiric function which gives, step by step, the next address to be inserted into the synthetic kernel reference stream. Finally, the distribution of the kernel-burst length and the distance between the beginning of two successive bursts are measured. The burst insertion may also be driven by information collected in the source traces, if the tracing tool records the system call positions. This allows us to generate more accurate workloads (e.g., to consider the fact that the processes typically exhibit a different number of system calls).

5.3.2 Validation of Kernel Insertion

This hybrid method introduces approximations concerning both the address generated and the distribution of kernel bursts. The weight of such error appears to be somewhat limited considering that the goal of the present analysis is the trace generation for performance evaluation of the memory

subsystem. To estimate the error induced by the synthetic generation of the kernel reference stream, a series of eight-processor traces distributed by Carnegie Mellon University and obtained by means of an Encore Multimax (shared-bus multiprocessor) machine (Table 5.1) has been considered. These traces represent a wide variety of application domains [Vashaw93]; they include both user and kernel references. A version of the Mach operating system from Carnegie Mellon University was used on this machine.

Table 5.1. The CMU multiprocessor traces.

Application	Source	Brief description
ecas	A. Wilson (Encore)	Computer Architecture Simulation
hartstone	N. Weiderman (SEI at CMU)	Real-Time Benchmark
locusroute	SPLASH (by J. Rose)	Circuit Routing
MP3D	SPLASH (by J.D. McDonald)	Rarefied Fluid Flow Simulation
ms_tracer	Fritzz Graphics (M. Rao at CMU)	Ray Tracing
pde	A. Wilson	Partial Differential Equation Solver

Table 5.2 includes the kernel access percentages (code, data, write) and the statistics concerning the distribution of distance and burst length. The latter are summarized by means of average value (μ) and standard deviation (σ).

Table 5.2. Kernel references statistics.

	Kernel	Kernel burst				Kernel code	Kernel data	
	references	distance		length		References	References	Writes
Application	(%)	μ	σ	μ	σ	(%)	(%)	(%)
ecas	3.32	27586	793	928	1288	2.12	1.21	0.45
hartstone	8.47	4004	9261	341	1421	5.36	3.11	1.05
locusroute	6.93	20214	12037	1404	2430	3.96	2.97	1.29
MP3D	3.21	28357	901	911	1134	2.05	1.17	0.43
ms_tracer	18.00	11045	18582	1581	13763	11.83	6.18	0.83
pde	5.30	21805	11369	1158	2189	3.40	1.90	0.75

Trace-driven simulation was used to compare the results of six different cases: i) the original CMU traces; ii) the CMU traces deprived of kernel references; iii) the original traces, with kernel references replaced by an address stream generated synthetically, yet preserving same position and length of each burst as in the original traces; iv) as in iii), with kernel-burst length and position generated stochastically by means of a distribution evaluated on a per-processor basis from the original traces; v) as in iv), with the distributions evaluated as average values over the entire set of processors; vi) the original traces, with kernel references generated synthetically for all traces on the basis of MP3D statistics, yet preserving the same position of each burst as in each original trace.

The following metrics representing the performance of a multiprocessor system are considered in this validation Section: Global System Power ($GSP = \sum U_{CPU}$, where $U_{CPU}=\frac{T_{cpu}-T_{delay}}{T_{cpu}}\times 100$ and T_{delay} is the total CPU delay time due to waiting for memory operation completions), average miss rate, bus utilization, and number of write transactions per thousand memory operations.

Table 5.3. Kernel model validation (case I: low bus utilization).

	Application	Actual	Error (%)				
		a	*b*	*c*	*d*	*e*	*f*
Global System Power	ecas	684.8	+3.6	−1.8	−1.8	−2.6	−1.1
	hartstone	780.4	+2.1	+0.1	+0.7	+0.2	−0.7
	locusroute	723.8	+7.2	−0.8	−0.2	−1.4	+4.0
	MP3D	632.1	+4.8	−0.1	−0.6	−0.7	−0.1
	pde	780.5	+1.5	+0.0	−0.1	−0.4	−0.2
	Average square error		4.4	0.9	0.9	1.4	1.9
Bus utilization (%)	ecas	65.3	−3.7	+3.4	+3.0	+4.2	+2.9
	hartstone	18.4	−72.8	−11.1	−17.7	−7.6	+6.0
	locusroute	44.6	−53.8	−2.7	−11.4	−8.1	−27.5
	MP3D	71.1	−6.7	+1.1	+0.8	+1.3	+1.1
	pde	23.1	−35.0	−1.7	−6.9	+3.4	−2.1
	Average square error		43.5	5.5	10.1	5.6	12.7

	Application	Actual	Error (%)				
		a	*b*	*c*	*d*	*e*	*f*
Miss rate	ecas	0.273	−17.5	+3.6	+4.0	+6.2	+1.5
	hartstone	0.071	−77.0	+14.0	+5.6	+23.9	+42.2
	locusroute	0.292	−61.3	+1.0	−8.6	−4.1	−29.8
	MP3D	0.522	−9.4	+1.1	+1.1	+1.5	+1.1
	pde	0.10	−40.0	+20.0	+20.0	+30.0	+24.0
	Average square error		48.2	11.1	9.1	17.5	25.5
Write transactions per 1000 memory operations	ecas	47.5	+6.3	+4.6	+3.7	+5.2	+5.0
	hartstone	10.58	−67.2	−19.2	−28.5	−17.0	+3.4
	locusroute	8.44	−19.9	−13.6	−15.1	−11.1	+11.9
	MP3D	0.88	−85.2	+23.8	+23.8	+37.5	+23.8
	pde	13.2	−27.2	−7.5	−14.4	−2.2	−21.0
	Average square error		51.0	15.5	19.1	19.2	15.4

Table 5.3 shows some results in terms of error percentages with respect to the values obtained from the actual traces. Columns *a* to *f* correspond to the six situations listed above, respectively. The simulations were performed for a 256-Kbyte, direct-mapped cache with a block size of 64 bytes and with the Dragon protocol. The timing parameters concerning the 64-bit bus are the same as in the example discussed in the Section 5.5 (Table 5.8). The sample trace that has been analyzed consists of 2,500,000 references per processor.

Although large errors affect both the miss rate and the number of bus transactions, a relatively low error can be observed in the GSP values, due to the fact that the system works with a low bus utilization ($\simeq$ 45 %). The percentage error concerning the GSP metric would drastically increase with larger values of bus utilization. Table 5.4 shows the results of a simulation in the case of a smaller cache size (64 KBytes), with an average bus utilization of 65.5%.

Table 5.4. Kernel model validation (case II: high bus utilization).

	Application	Actual	Error (%)				
		a	*b*	*c*	*d*	*e*	*f*
Global System Power	ecas	357.1	+9.6	+ 3.1	+3.1	+0.0	+3.4
	hartstone	765.7	+3.5	+0.2	+0.9	+0.9	−1.2
	locusroute	555.6	+30.2	+6.3	+9.1	+6.4	+20.2
	MP3D	420.4	+15.0	+3.7	+3.6	+3.2	+3.7
	pde	740.1	+4.7	+0.3	+1.3	+0.6	+1.3
	Average square error		15.9	3.5	4.6	3.2	9.3
Bus utilization (%)	ecas	95.1	−2.0	−0.7	−0.7	−0.1	−0.6
	hartstone	23.9	−71.1	−13.1	−17.1	−16.8	+9.0
	locusroute	81.6	−51.5	−11.6	−17.9	−15.4	−31.3
	MP3D	94.7	−4.1	−1.0	−0.9	−0.8	−1.0
	pde	37.2	−39.1	−3.5	−12.6	−4.2	−10.2
	Average square error		43.0	8.0	12.4	10.4	15.3

	Application	Actual	Error (%)				
		a	*b*	*c*	*d*	*e*	*f*
Miss rate	ecas	1.247	−6.3	−3.4	−4.0	+0.5	−3.8
	hartstone	0.130	−77.7	−4.6	−10.7	−10.0	+28.4
	locusroute	0.768	−63.3	−22.6	−29.9	−26.3	−45.8
	MP3D	0.891	−15.0	−4.9	−4.7	−4.5	−4.9
	pde	0.225	−45.8	−0.8	−8.9	−0.8	−6.2
	Average square error		49.8	10.6	15.0	12.8	24.4
Write transactions per 1000 memory operations	ecas	35.63	+1.4	−0.1	+0.0	−0.7	−0.0
	hartstone	8.43	−58.8	−28.8	−35.8	−23.6	−19.0
	locusroute	4.63	−36.5	−32.2	−20.7	−34.5	+27.9
	MP3D	0.67	−81.6	−37.6	−36.7	−14.2	−37.6
	pde	9.75	−16.7	−7.5	−17.6	+2.7	−27.0
	Average square error		48.4	25.8	25.9	19.8	25.6

In any case, the insertion of a stochastically generated kernel reference stream reduces the error (compared to the complete elimination of kernel references – column *b*), and the reduction is far more appreciable if detailed information concerning kernel bursts is available. This appears quite evident from a comparison of columns *d* and *e* vs. column *c*. The same comparison, however, shows that it is not generally worth reproducing exactly the original position of kernel accesses within the

global reference stream in order to improve the accuracy of the model. Finally, the accuracy of the proposed methodology concerning its effects on the memory subsystem also depends on the behavioral differences between the workload from which the kernel statistics are extracted and the applications to which the synthetic kernel generation model is applied (column *f* in Tables 5.3 and 5.4).

5.3.3 Process Management and Virtual-To-Physical Address Translation

One of the main goals of the multiprocessor scheduler is to provide an acceptable degree of load balance in order to allow the programmer to develop his applications without caring about the load distribution on the processors. Nevertheless, load balance induces process migration that causes further coherence overhead. Actually, a memory block belonging to a private area of a process can be replicated in more than one cache as a consequence of the migration of the process which owns this block. These copies have to be treated as shared with respect to the coherence-related operations, resulting in a heavy and useless burden for the shared bus (*passive sharing* [Prete97], *process-migration sharing* [Hwang93]). Furthermore, on every context switch, a burst of cache misses occurs, due to the loading of the working set of the new process. A scheduling policy based on cache affinity [Squillante93] can reduce the effects of both issues just mentioned.

Trace Factory models the process management aspects by simulating a simple scheduler. The input parameters for the scheduler are: the number of processes (N_{proc}), the number of processors of the target machine (N_{cpu}), the time slice in terms of number of references (T_{slice}), the process activation policy (*two-phase* or *non-blocking*), and the process scheduling policy (*random* or *affinity*). Trace Factory simulates the scheduler in the following way: i) it starts from a set of source traces, one trace for each uniprocess application and as many traces as the number of processes belonging to the multiprocess application, and ii) produces as many target traces as the number of processors of the target machine. The whole scheduling activity can be directly driven by the simulator; in this case, the scheduling activity is conditioned by the speed of each simulated processor.

The scheduler operates as follows: if a process *p* is running on processor *P* for a D time interval (again specified in terms of number of references) then *D* references of the *p* source trace become references for processor *P*. At the simulation start up, all the processes are ready and they are inserted in a proper queue, namely R_1. Initially, the scheduler randomly selects N_{cpu} processes, and each running process has a different time slice (namely, the process running on processor i is assigned a time slice $T_i = \frac{i \cdot T_{slice}}{N_{cpu}}$). After the first context switch on each processor the next scheduled process is regularly assigned T_{slice}. This strategy, typically adopted in operating systems

for multiprocessors, avoids a context switch being simultaneously needed on each processor every T_{slice}, which would produce an undesirable overlap of miss peaks on all caches and a consequent bus saturation due to the bus transactions needed to fetch missing blocks from memory.

On a context switch, a process is extracted from R_1 and assigned to the available processor. The choice of such process can be made either according to the cache affinity policy mentioned above, or just randomly. The preempted process may be managed in two different ways. In the *non-blocking* activation policy, the preempted process is immediately inserted into the R_1 queue. This strategy suffers from the starvation problem: this implies that references of a process may be not present within a target trace, when its length is short and $\frac{N_{proc}}{N_{cpu}} \gg 1$. A second activation policy (*two-phase*) makes use of another queue, namely R_2, initially empty (Figure 5.2). On every context switch, the preempted process is inserted into R_2 (phase one). As soon as the queue R_1 becomes empty, all the processes are taken from R_2 and inserted into R_1 (phase two). This technique ensures that a process does not have to wait an indefinite time for its turn: indeed, a process cannot be executed $n + 1$ times before each other process is executed exactly n times.

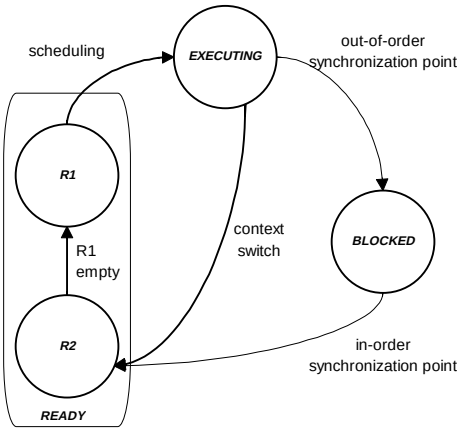

Figure 5.2. State transition diagram in the case of two-phase activation strategy.

Finally, the scheduler can consider the synchronization sequence produced by a multiprocess application execution. In this case, the process scheduling is driven by the time slice for processes belonging to uniprocess applications, and by both the time slice and the synchronization sequence for multiprocess applications. Source traces have to include synchronization tags representing the actual synchronization sequence of the parallel application execution [Vashaw93]. When a process reaches an out-of-order synchronization event, it is inserted into a waiting queue to wait for the synchronization event. Then, it enters either the R_1 or the R_2 queue as described above.

In virtual memory models based on paging, the localities of virtual and physical references produced by a running process may be different. The mapping of sequential virtual pages into non-

sequential physical pages causes this difference and influences the number of *intrinsic interference* (or *capacity*) *misses* due to interferences among kernel code and data, user data and code accesses within the same cache set [Hennessy96].

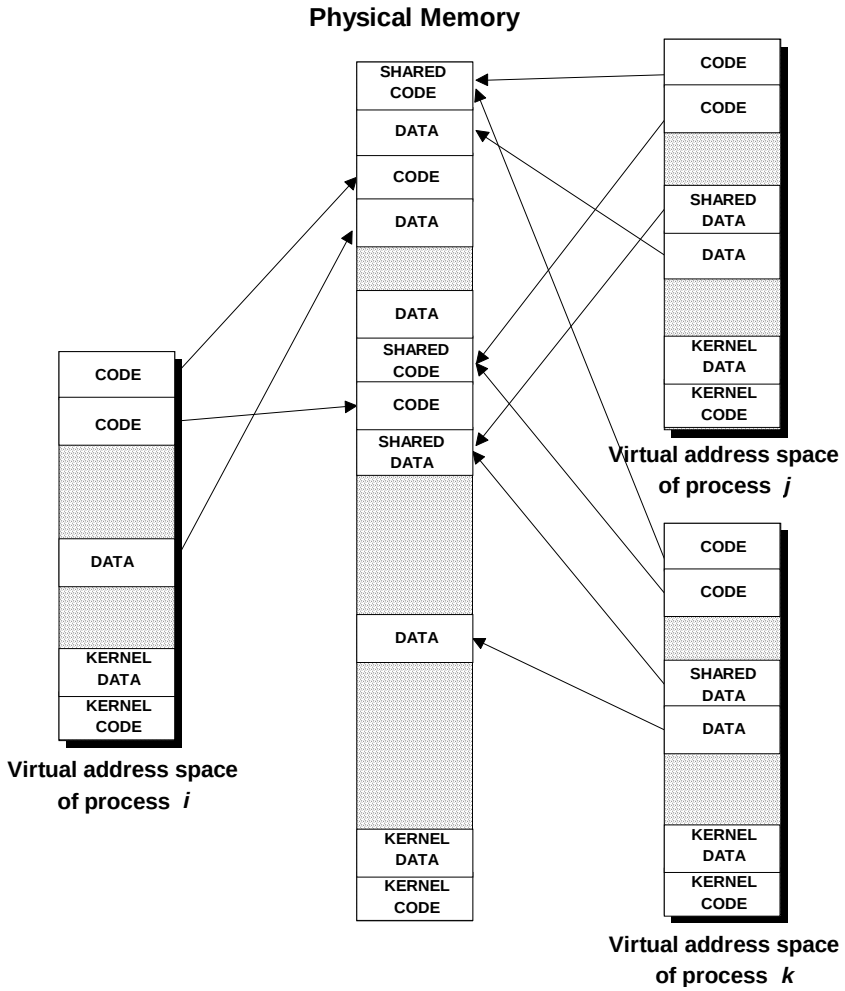

Figure 5.3. A scheme of the virtual-to-physical address translation.

The virtual-to-physical address translation is modeled as follows. Each process has its own address space for code and private data. The kernel and its associated data structures reside at the top of the virtual address space of each process. When a set of processes share a memory area, the system ensures that for such processes, the shared areas are mapped on the same set of physical memory pages. When a number of instances of the same application are active, their code is shared. Finally, kernel instances share a unique set of physical memory pages. The page size and the physical memory size are input parameters for the address translation mechanism.

Each process starts its execution without any page being stored into physical memory in advance.

As soon as it tries to fetch a location within a page which is not resident in the physical memory, a page fault is generated and the pertinent page is allocated (*on demand* paging). As a consequence of this page fault, a context switch is generated, the process is suspended from execution and spends a predefined number of cycles in the waiting queue. In this way; the delay needed to fetch the required page from disk has been modeled. Since the traces lack any information concerning I/O operations, no other aspect of I/O interaction is considered.

5.4 Workload characteristics

In the following, the goal is to evaluate and compare coherence protocol performance on a general-purpose multiprocessor workstation. Thus, the previous technique has been used to generate three nontrivial real workloads, named Const–30, CVar50–30, and MVar50–30. Each workload includes 60 million references.

Table 5.5. Statistics of uniprocess application and Unix command traces (64-byte block size, 2,500,000 references).

Application	Distinct Blocks	Code (%)	Data (%)		System Calls
			Read	Write	
awk (beg)	4963	76.76	14.76	8.47	29
awk (mid)	3832	76.59	14.48	8.93	47
cjpeg	1803	81.35	13.01	5.64	18
cp (beg)	2615	77.53	13.87	8.60	26526
cp (mid)	2039	78.60	14.17	7.23	56388
msim	960	84.51	10.48	5.01	345
dd	139	77.47	16.28	6.25	47821
djpeg (beg)	2013	81.00	12.75	6.26	15
du	1190	75.86	16.37	7.77	9474
lex	2126	78.67	15.49	5.84	40
gzip	3518	82.84	14.88	2.28	13
ls -aR	2911	80.62	13.84	5.54	1196
ls -aR	2911	80.62	13.84	5.54	1196
ls -ltR (beg)	2798	78.77	14.58	6.64	1321
ls -ltR (mid)	2436	78.42	14.07	7.51	1778
rm (beg)	1314	86.39	11.51	2.10	10259
rm (mid)	1013	86.29	11.65	2.06	15716
telnet (beg)	781	82.52	13.17	4.31	2401
telnet (mid)	205	82.78	12.93	4.28	2827

Const–30 consists of 30 typical sequential programs such as system commands, utilities, and user applications. Some typical Unix commands have been selected (awk, cp, dd, du, lex, rm, and ls) with different command-line options, three utility programs (cjpeg, djpeg, and gzip), a network

application (telnet), and a user application (msim, the multiprocessor simulator used in this work). In a typical situation, various users may run different system commands and ordinary applications. To take into account that users can launch the same program at different times, some commands are traced in shifted execution sections: initial (*beg*) and middle (*mid*). Table 5.5 shows these source traces in terms of: i) number of distinct (unique) blocks the program uses; ii) code, data-read, and data-write access percentages; iii) number of system calls.

In CVar50–30 and MVar50–30, a parallel application is added to the basic Const–30 workload. The parallel application generates a number of processes equal to 50% of the machine processors. Since the access pattern to shared data of parallel applications influences multiprocessor performance significantly, two parallel programs with different sharing behavior, MP3D and Cholesky (both from the SPLASH suite [Singh92]), have been considered. MP3D simulates rarefied hypersonic flow; the generated trace relates to a case of 10,000 molecules and 20 time steps. Cholesky factorizes a sparse positive definite matrix, using the homonymous method. For Cholesky, the trace has been generated by using a 1,806x1,806 matrix with 30,284 nonzero elements coming from the Boing/Harwell sparse matrix test (bcsttk14) as input.

Table 5.6 summarizes the multiprocess-application trace statistics. The write-run figures show that: i) MP3D exhibits coarse-grained sharing, since the average write-run length varies from 5.89 to 8.18; and ii) Cholesky exhibits medium-grained sharing, having an average write-run length from 4.75 to 5.10.

Table 5.6. Statistics of multiprocess application source traces (MP3D and Cholesky) in case of 64-byte block and 60,000,000 references. In particular, the table shows the mean value (μ) and standard deviation (σ) of WRL and XRR statistics.

Workload	# of Tasks	Total Distinct Blocks	Code (%)	Data		Shared Distinct Blocks	Shared Data		Write-run			
				Read (%)	Write (%)		Acc. (%)	Write (%)	WRL		XRR	
									μ	σ	μ	σ
MP3D	4	6480	78.56	14.30	7.14	1625	10.34	3.20	8.18	6.04	1.54	1.60
	6	6923	78.70	13.99	7.31	2004	10.91	3.56	7.03	5.06	1.51	1.63
	8	7169	78.77	13.84	7.39	2309	11.30	3.76	6.55	4.65	1.50	1.67
	10	7308	78.81	13.74	7.45	2597	11.62	3.91	6.25	4.40	1.50	1.71
	12	7397	78.83	13.68	7.49	2820	11.88	4.02	6.07	4.32	1.50	1.73
	14	7509	78.85	13.65	7.50	3002	12.07	4.10	5.89	4.18	1.51	1.75
Cholesky	4	17119	79.83	13.57	6.60	7215	8.29	1.19	4.75	3.47	1.06	0.65
	6	19172	80.21	13.65	6.14	8789	9.67	1.32	4.46	3.12	1.06	0.68
	8	20569	80.43	13.66	5.91	10079	10.24	1.36	4.43	3.04	1.05	0.63
	10	21557	80.69	13.70	5.61	11268	10.82	1.44	4.54	3.00	1.06	0.74
	12	22900	80.98	13.69	5.33	12404	11.18	1.44	4.89	3.67	1.05	0.61
	14	23669	81.11	13.70	5.19	12876	11.47	1.48	5.10	3.80	1.05	0.63

The target trace characteristics resulting from a simulation performed in the reference case study

Table 5.7. Statistics of target traces produced to evaluate Dragon protocol in case of 64-byte block and 60,000,000 references. In particular, the table shows the mean value (μ) and standard deviation (σ) of WRL and XRR statistics. "Par. App." is the percentage of references belonging to the parallel application

Workload	PEs	Par. App. (%)	Total Distinct Blocks	Code (%)	Data		Shared Distinct Blocks	Shared Data		Write-run			
					Read (%)	Write (%)		Acc. (%)	Write (%)	WRL		XRR	
										μ	σ	μ	σ
Const–30	8	0	65989	77.00	13.52	9.48	9912	16.52	4.96	20.63	10.48	5.30	8.84
	12	0	97218	77.10	13.26	9.64	13577	17.24	5.32	20.05	9.83	6.32	9.30
	16	0	125186	77.10	13.24	9.66	15692	17.57	5.44	19.69	10.64	5.82	9.04
	20	0	183987	77.28	13.07	9.65	17249	17.63	5.45	19.14	11.21	5.76	9.01
	24	0	257916	77.52	12.80	9.68	19954	17.32	5.40	18.54	11.19	5.09	8.40
CVar50–30	8	12.27	82096	77.14	13.35	9.51	11102	16.40	5.93	15.50	10.10	4.01	6.86
	12	17.92	85168	76.81	13.37	9.82	17003	17.55	5.45	12.24	9.16	3.40	5.45
	16	22.24	105771	76.86	13.25	9.89	20876	17.96	5.74	10.11	8.16	2.95	4.75
	20	26.83	122748	76.90	13.28	9.82	21129	18.29	5.88	9.15	7.43	2.74	4.34
	24	30.58	155872	76.83	13.19	9.98	22544	18.42	5.94	8.49	6.97	2.57	3.94
MVar50–30	8	11.76	85702	77.28	13.23	9.49	12124	16.26	4.92	15.42	10.93	4.38	7.52
	12	16.67	91582	76.99	13.22	9.79	17886	17.04	5.31	12.56	10.71	3.34	6.51
	16	21.05	112150	77.12	13.24	9.64	22196	17.41	5.38	10.19	10.15	2.80	5.85
	20	25.00	131536	77.10	13.24	9.66	24574	17.66	5.45	9.95	9.89	2.53	5.57
	24	28.57	166810	77.27	13.20	9.53	27058	17.77	5.42	9.99	9.91	2.42	5.34

(Section 6.1) are summarized in Table 5.7. Comparing the write-run statistics of Tables 5.6 and 5.7, it can be observed that the write-run of target traces results quite high, even much higher than in source traces. This is due to process migration. Therefore, this aspect strongly motivates the introduction of kernel modeling in the evaluation of such multiprocessors with this kind of workload.

5.5 Multiprocessor Simulator

The multiprocessor simulator [Prete95] used in this analysis characterizes a shared-bus multithreaded multiprocessor in terms of CPU, cache, and bus parameters. This simulator models simple multithreaded processor architecture. The target of this evaluation is to show how the memory hierarchy is influenced by the choice of an adequate coherence protocols. Some memory-latency hiding techniques is not modeled since they do not modify coherence protocol behavior, and that are currently used in modern processors.

5.5.1 Simulator Input Parameters

The CPU parameters are the clock cycle, the minimal number of clock cycles for a read/write operation, and the temporal distribution of the memory accesses. This distribution is described in terms of the maximum number (*M*) of references per *time interval* and the probability that this interval contains exactly 0, 1, 2,. . . , *M* memory references. That time interval is a fixed number of CPU clock cycles.

Both the number of processors and the number of context per processor can be chosen arbitrarily.

The cache parameters are cache size, block size, associativity, and the access time for read/write operation. The cache block replacement policy is LRU (Least Recently Used). The simulator models a multiprocessor having a relaxed memory consistency (processor consistency [Gharachorloo90], [Adve96], [Hennessy96], [Milutinovic98]). This is implemented allowing the write transaction buffering.

Finally, the bus parameters are the bus width and the number of CPU clock cycles for each kind of transaction: write, invalidation, update-block, and memory and cache read-block. Table 5.8 reports the values of CPU, cache, and bus parameters for the reference case study (Section 6.1). The main memory is supposed to support write buffering. In this way, the cost of write transaction is equal to the cost of invalidation transaction.

Table 5.8. Numerical values of the reference simulated system (timings are specified in terms of clock cycles).

Class	Parameter	Timings
CPU	Read cycle	2
	Write cycle	2
	Duration of each slice (cycles)	4
	Maximum number of references per slice	2
	Probability of [0,1,2] references per slice	{0.1,0.3,0.6}
	Number of Contexts	1
Cache	Cache size	256 kbytes
	Block size	64 bytes
	State updating	1
	Write cycle	1
	Read cycle	1
Bus	Width	64 bit
	Write transaction	5
	Invalidate transaction	5
	Update-Block transaction	32
	Memory Read-Block transaction	24
	Cache Read-Block transaction	18

5.5.2 Simulator Output Parameters

The simulator can generate a number of statistical values, such as: miss ratio, number of write transactions invalidation, update-block, and memory and cache read-block per memory operation, and bus utilization ratio. In the discussion below, the focus is on the Global System Power metric (GSP) [Archibald86] that represents the number of the processors of an ideal machine that does not

have delay in accessing memory:

$$GSP = \sum U_{cpu}$$

where

$$U_{cpu} = \frac{(T_{cpu} - T_{delay})}{T_{cpu}}$$

T_{cpu} is the time needed to execute the workload, and T_{delay} is the total CPU delay time due to waiting for memory operation completion. Instead of execution time, this metric has been used since in the below experiments the target machine does not execute a single program, but a combination of portions of programs. At the same time, the workload characteristics (such as the number of processes of the parallel application) change as the number of processors changes. In this condition, GSP gives the necessary comparability when the performance evaluation requires varying the number of processors and other system parameters.

Another metric to show the effectiveness of each protocol in achieving coherence at reduced traffic overhead, is Processor/Bus Efficiency (PBE):

$$PBE = GSP/BUR$$

where the BUR is the bus utilization ratio and its value ranges between 0 and 1. A high value of this number indicates that a protocol is exploiting the bus more effectively to get a given value of GSP. For example, if two protocols have the same value of GSP, but have different bus utilization ratios, then the protocol having the higher PBE is using the bus bandwidth more effectively.

All the statistics regarding single processor performance are averaged over the total number of processors.

Chapter 6

Performance Evaluation

In this chapter, the effectiveness of PSCR protocol under various architectural parameters and scheduling policies will be shown.

As a reference case study a 64-bit bus, 64-byte cache block size, direct-mapped cache, single-threaded multiprocessor will be analyzed considering the three real workload previously introduced: Const–30, CVar50–30, and MVar50–30.

Successively, the parameter space will be explored. Explicit simulations have been carried out for different cache block sizes, different cache associativities, different cache sizes, different bus-widths, different scheduling policies, different processor speeds, different number of contexts per processor, thus covering a quite widespread range of possibilities for building a multithreaded multiprocessor that could be used as a general purpose workstation.

The goal is to show the effectiveness of PSCR, in all the above situations. PSCR performance is compared against six other protocols: Dragon, Berkeley, MESI, Competitive Snooping, Update-Once, and AMSD. These protocols belong to different classes, that is they use different strategies to obtain cache coherency. The selection is motivated as follows:

- In the WU class, the most widely used protocols are Dragon and Firefly, however, Dragon is usually found to perform slightly better.
- In the WI class, two protocols have been evaluated: MESI, the most widely used protocol, and Berkeley, which is more often evaluated in the literature.
- In the HY class, Competitive Snooping and Update-Once have a good performance over a wide range of applications and employ different strategies to switch from WU to WI.
- In the AH class, the most promising protocol is AMSD to treat passive sharing.

6.1 Reference Case Study

As a starting point of this evaluation, and for an in-depth discussion, the case study of a machine having a 64-bit data bus width, and 64-byte cache block size has been chosen. Each processor has a 256-Kbyte, direct access private cache, and 1 context per processor. This case study is generic and representative of the various simulated instances (Figure 6.1). In Table 5.8, CPU, cache, and bus parameter values are reported.

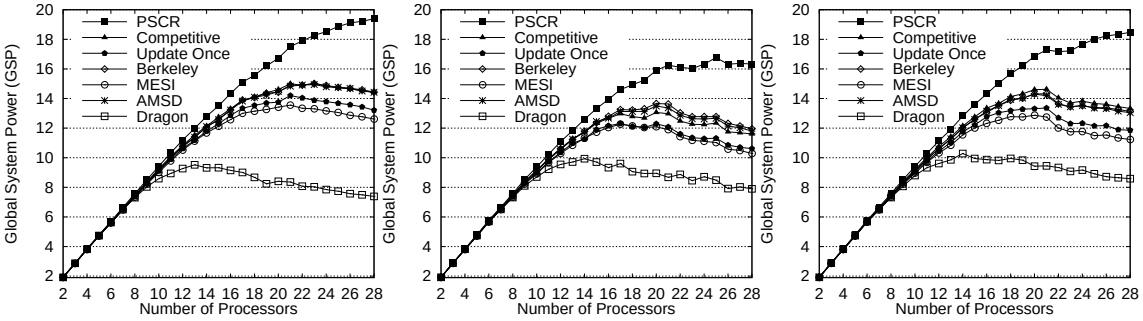

Figure 6.1. Reference case study (64-bit bus, 64-byte block size): Global System Power. GSP of PSCR is at least 40% higher than the other protocols' GSP.

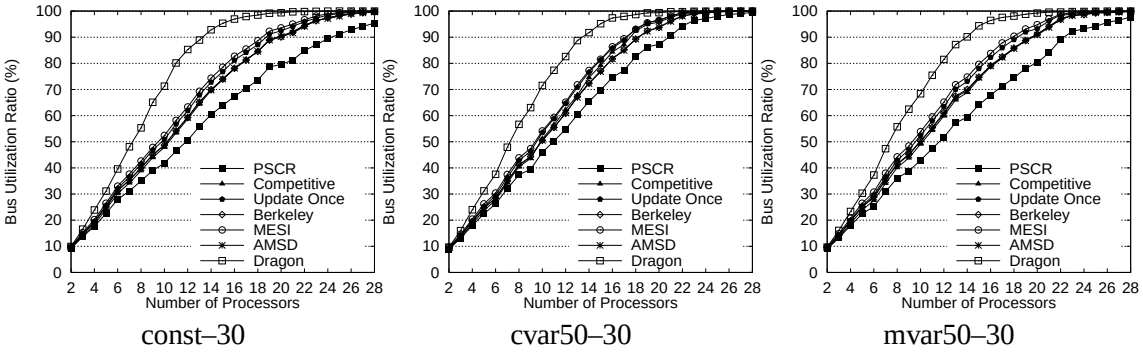

Figure 6.2. Reference case study (64-bit bus, 64-byte block size): Bus Utilization Ratio. PSCR takes advantage of the reduced bus traffic.

The simulations, related to the three workloads described above, yield the following results for the Global System Power. GSP for PSCR is at least 40% higher than that of other protocols' GSP. Moreover, PSCR scales up better, making it possible to connect more processors on the same bus. As expected Dragon has the worst behavior in terms of both absolute performance (GSP) and scalability. Excluding PSCR, AMSD is the best performing protocol. Its behavior is near or better

than all the four other protocols (Competitive Snooping, Update-Once, Berkeley, and MESI), for all workloads. The behavior of these four protocols exhibits workload sensitivity.

The good performance of PSCR is mainly due to the lower number of bus transactions, and hence, lower global traffic on the shared bus, which is the bottleneck of the system (Figure 6.2). The reduced bus traffic minimizes the latency of processor operations.

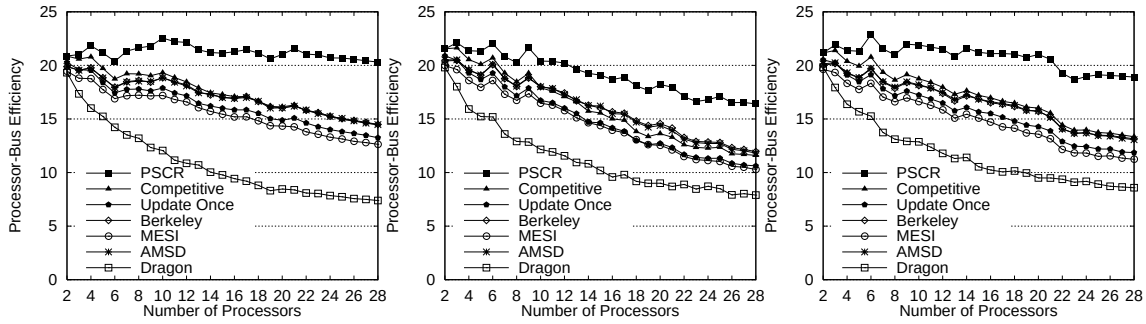

Figure 6.3. Reference case study (64-bit bus, 64-byte block size): Processor/Bus Efficiency. PSCR exploits better the available bus bandwidth. This enhances the protocol scalability.

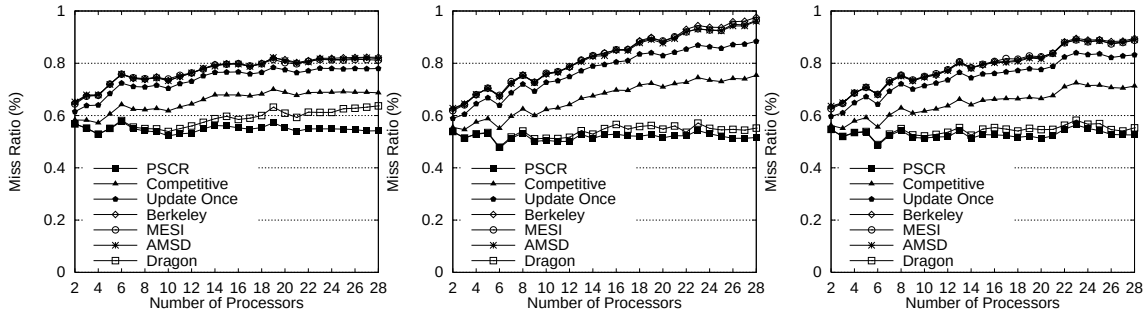

Figure 6.4. Reference case study (64-bit bus, 64-byte block size): Miss rate. PSCR exhibits the smaller miss ratio. The invalidation strategy of Update Once, Berkeley, MESI, and AMSD causes a miss ratio increase for those workloads including a parallel application. Competitive Snoopy Caching has an intermediate behavior.

In addition, PSCR exploits the available bus bandwidth better, that is, for a given percentage of bus utilization, PSCR delivers a higher GSP in comparison to all other protocols (Figure 6.3). This is also evident by Dragon behavior and PSCR bus utilization. For example in Figure 6.3 for the Const–30 workload, Dragon reaches 80% for 11 processors and starts to saturate (Figure 6.1). Instead, PSCR reaches 80% for 20 processors, but it still has the possibility to scale up. This two

advantages (reduced global traffic and better usage of the available bus bandwidth) of PSCR can be further explained by detailing the quantity and type of bus transactions actually needed by each protocol. This is showed in the following graphs (Figure 6.4-6.7), and discussed below.

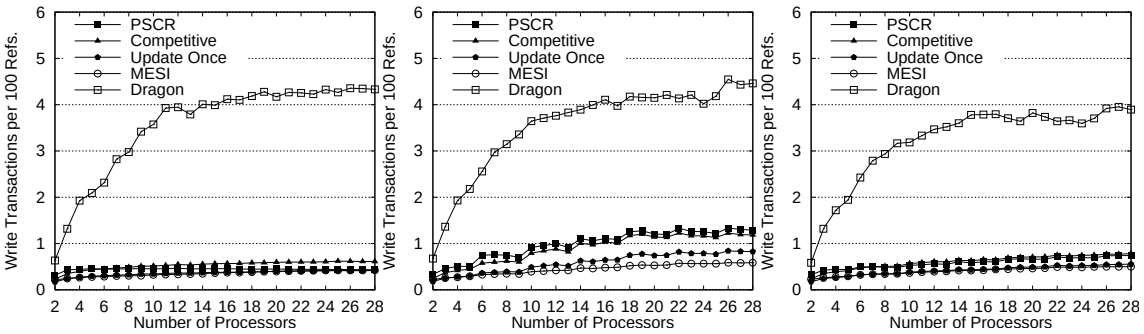

Figure 6.5. Reference case study (64-bit bus, 64-byte block size): number of Write transactions per 100 references. We represent data relative only to those protocols having write transactions. The introduction of a parallel application (CVar50–30 and MVar50–30) increases the number of write transactions. In the case of Dragon, this does not happen since this protocol is more sensitive to passive-sharing effects, which increase as the number of processes (for a given number of processors).

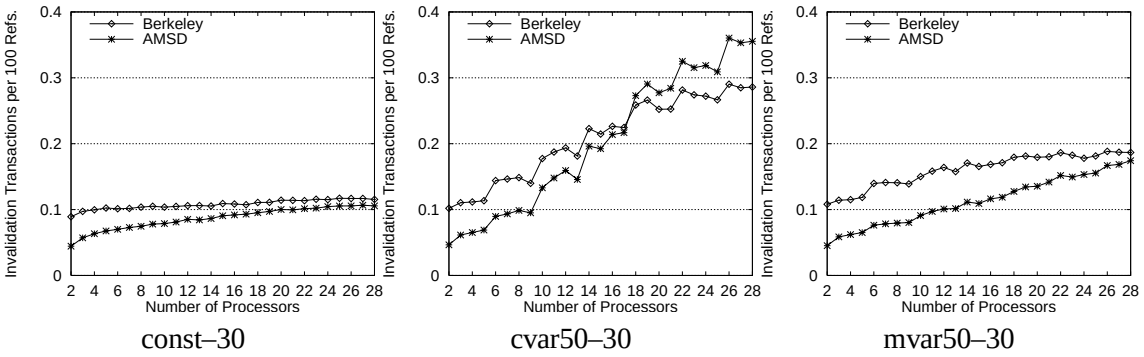

Figure 6.6. Reference case study (64-bit bus, 64-byte block size): number of Invalidate transactions per 100 references for Berkeley and AMSD protocols. Parallel applications sharing increases the number of invalidations for Berkeley and AMSD.

The reduction of coherency-related operations (write and invalidate transactions) results in a real advantage, only if the number of read-block transactions does not increase due to a higher miss rate.

Miss and write handling introduce a different bus cost and latency for the processors. Miss operations due to read operations may introduce a delay, when the processor or other units have to

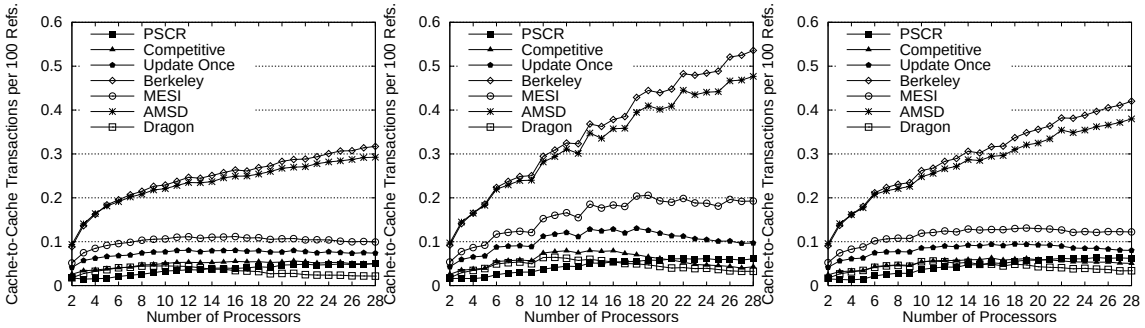

Figure 6.7. Reference case study (64-bit bus, 64-byte block size): number of cache read-block transactions. AMSD and Berkeley heavily employ this cheaper transaction.

wait for the operation termination in order to continue the execution. As for the write operations (in both hit/miss conditions), they can be managed in an asynchronous way because of write buffering, i.e. the processor can start working on the next operation even though the current one has not been actually completed. This implies that write operations do not involve idle time for the CPU directly, although they may significantly affect miss cost, because they cause more bus traffic. Moreover, it has to be considered that the amount of data to be transferred on the shared bus is significantly lower in case of write transaction than in case of read-block transaction.

PSCR protocol has both a low miss rate (Figure 6.4) and a limited number of write transactions (see Figure6.5). These two aspects are strongly correlated, in that PSCR selective invalidation strategy only eliminates the useless copies of P-blocks, without causing further unnecessary misses.

Berkeley, AMSD, MESI, and Update-Once exhibit a lower number of coherence-related bus operations (invalidations in some cases, writes in others) than PSCR, but at the cost of an increased miss rate. This effect is due to the invalidation strategy. In case of Berkeley and AMSD, the invalidation strategy causes a miss increase, whose consequences are limited because these protocols heavily employ the cheaper cache read-block transaction (Figure 6.7). This fact explains the performance differences among Berkeley, AMSD, MESI, and Update-Once that have similar values of miss ratio. Competitive Snooping has a low number of write transactions, like PSCR, and a good behavior as for the misses, thus it has an intermediate behavior between PSCR and all the other protocols (except Dragon). Dragon is greatly penalized by the high number of write transactions.

The introduction of parallel applications (CVar50–30 and MVar50–30 workloads) penalizes the global performance of each protocol (except Dragon), because of the overhead required to keep the active shared copies coherent (Figures 6.1, 6.4, 6.6, and 6.7). The differences in terms of write per-

centage and write-run length between CVar50–30 and MVar50–30 cause different overhead (Figures 6.4, 6.5, and 6.6) and performance (Figure 6.1) for all protocols (except Competitive Snooping). In the case of Dragon, this phenomenon is not observable, because of the saturation of the bus, which starts from a low number of processors. On the contrary, the reuse of active shared copies causes a GSP increase for Dragon in case of CVar50–30 and MVar50–30.

6.2 Influence of Cache Structure

In the second step of this analysis, the behavior of the protocols in the case of a 2-way and a 4-way set-associative cache is examined. The other parameters have the same values as in the previous case. For all protocols, the simulation yielded the following results:

- a decrease of the miss rate (and, as a consequence, the number of read-block transactions);
- an increase of the number of write or invalidate transactions.

The miss rate decrease is essentially due to the higher associativity, which offers more caching alternatives for blocks sharing the same cache set. The increase of the number of write or invalidate transactions is caused by the high number of shared copies, in turn due to the longer lifetime of cached blocks. PSCR exhibits the highest increment in the GSP values, as the associativity increases. This can be explained as follows.

Bus traffic results from the sum of two components: i) number of read-block transactions caused by miss conditions, and ii) number of coherence actions (write and invalidate transactions).

Misses have three independent sources: misses due to newly accessed blocks, capacity misses, and invalidation misses. The higher associativity causes a reduction of capacity misses and consequently enhances the effects of coherence-related bus traffic on global performance. Furthermore, the increase of associativity generally produces an increase of coherence-related activity, since a larger number of shared copies can be involved in write operations. For this reason, the protocols that generate the lowest total number of coherence-related bus actions yield higher GSP.

The complete set of graphs is not reported, since they are qualitatively similar to those obtained in the reference case study. To summarize the results, it can be observed from GSP graphs, that protocols exhibit a good scalability until the system reaches a critical number of processors. For each protocol, beyond this critical point does not make sense to attach more processors on the bus. A quantitative estimation of this critical point can be given by defining it as the point in which GSP graph slope is 70% of the initial slope. Figure 6.8 shows the critical point for direct-access (reference case) and 2-way set-associative cache.

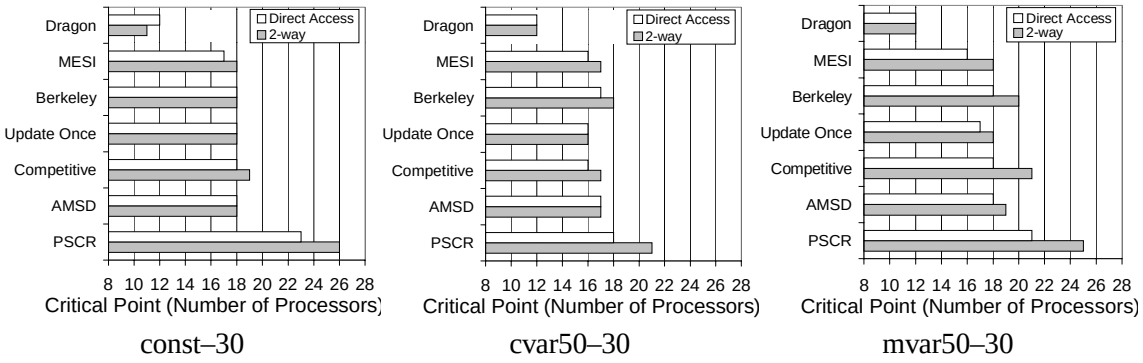

Figure 6.8. The critical point for direct-access and 2-way set-associative cache. PSCR has the highest GSP as the associativity increases.

Figure 6.9 shows the GSP of each protocol at the critical point in the case of direct-access, 2-way and 4-way set-associative caches. Generally, both scalability and processing power furnished by the machine greatly increase as switching from one to two ways. This increment is limited in case of two- and four-way caches.

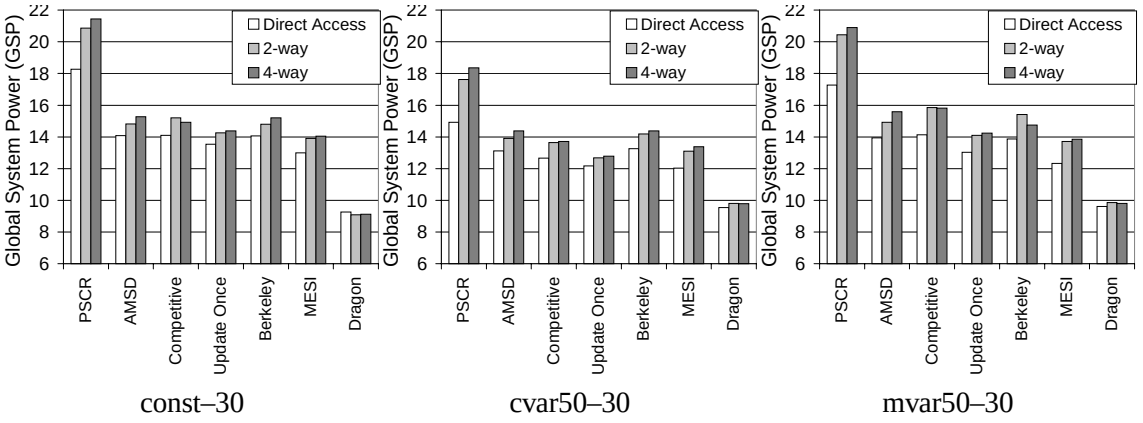

Figure 6.9. The GSP of PSCR against all other protocols, at the critical point for direct-access, 2-way, 4-way caches. For each workload and protocol, the GSP at the critical point is reported. PSCR exhibits the highest performance increase than the other protocols, as the number of ways increases. Dragon has a behavior quite independent of the number of ways.

The performance of 64, 128, 256 and 512-Kbyte, 2-way caches has been analyzed. In all the experiments, PSCR had the best performance, compared with the other protocols. The performance slightly increases for all the protocols with the cache size, in a way similar to the case of increased associativity. The results show a decrease of the miss rate and an increase of the number of shared copies, due to the longer lifetime of cached blocks.

As the cache size is decreased, Dragon exhibits a small increase in its performance. For a 64-Kbyte cache, its performance is also comparable with WI protocols, especially when true sharing is higher (CVar50–30 and MVar50–30 workloads). As said above, and known from the literature, this is due to the high update traffic of Dragon. For smaller cache sizes the copy eviction due to the limited cache size, is equivalent to a mechanism of copy invalidation that eliminates the remote copies, which are not accessed for long time intervals (i.e. that exhibit a long WRL). Finally, the performance as the block size is varied (Figure 6.10) has been analyzed. For Const–30 workload, PSCR exhibits a slight GSP improvement as the block size increases from 64 bytes to 128 bytes. For a 256-byte block size, GSP values are slightly lower than in the 64-byte case. Dragon's GSP has the same values in the 64 and 128-byte case, whilst there is a slight decrease in the 256-byte case. WI protocols are clearly penalized in the 256-byte case, reducing their performance at Dragon level. Competitive Snooping does better than WI protocols in case of 128 and 256-byte block size. In case of CVar50–30 and MVar50–30 workloads, WI protocols becomes even worse than the Dragon one.

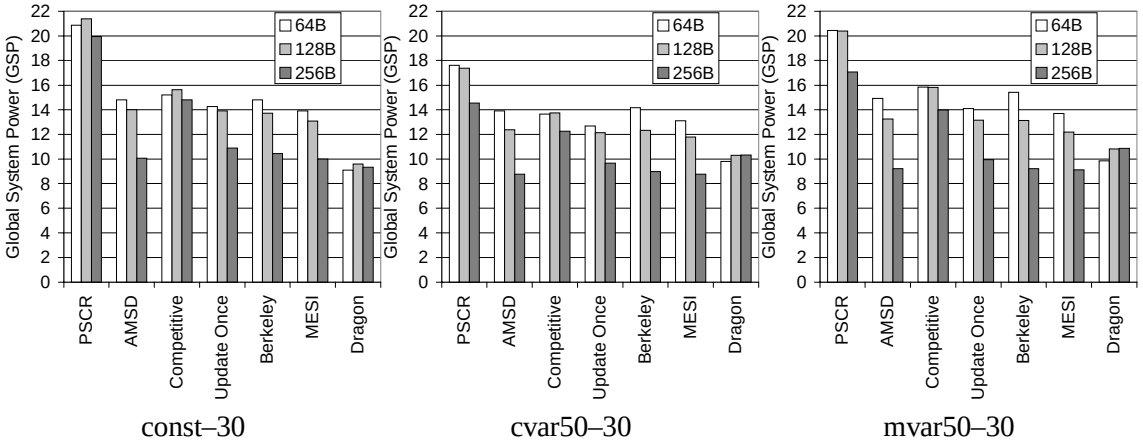

Figure 6.10. The GSP of PSCR against all other protocols, at the critical point for 2-way set associative caches with 64, 128 and 256-byte block size). PSCR is always the best performing protocol. This is due to the Write-Update strategy on S-blocks. Dragon has a behavior quite independent of the block size.

Indeed, the bigger block size naturally causes a miss ratio decrease, but a longer time to handle the miss itself. In case of Dragon and PSCR protocols, the latter two phenomena have opposite but balanced weigh on the performance (64 and 128-byte case). For the 256-byte case the even longer time to load the block becomes more weighing on GSP. WI protocols are greatly penalized by the higher miss cost due to the bigger block size. Competitive Snooping adapts its behavior based on the ratio between miss and write cost, thus achieving better performance than WI protocols.

6.3 Influence of the memory latency

The value of memory latency considered in the evaluations just exposed is somewhat low relative to current and future machines [Gee93]. For this reason, a system with larger memory latency (30 cycles, against the 6 cycles for a memory access in the reference case) has been considered. In this case, for all protocols, it has been found that a 128-byte block size yields a better performance than a 64-byte one, due to the higher cost of memory access. New bus timings are shown in Table 6.3.

Table 6.1. Bus timings (cycles) for the high-latency case study (64-bit bus, 128-byte block size).

READ-BLOCK		WRITE	INVALIDATE	UPDATE-BLOCK
MEMORY	CACHE			
96	40	5	5	34

The simulation results are presented in Figure 6.11. Each protocol reaches the critical point for a lower number of processors, in respect to the previous situations. Even in this situation, PSCR exhibits the best performance for all the considered workloads. Dragon and Competitive Snooping take advantage of this situation. The higher miss cost penalizes the protocols based on non-selective invalidation. The write cost is constant. For this reason, good results are obtained by Dragon (which does not invalidate), Competitive Snooping (which takes into account the cost ratio of read-block to write transaction in making decisions about invalidating) and PSCR (which only invalidates private data copies). The good behavior of Dragon is obtained only in the case of CVar50–30 and MVar50–30 workloads. This is due to the presence of parallel workload, which allows Dragon to reuse the shared copies, thus obtaining a miss ratio decrease (Figure 6.4).

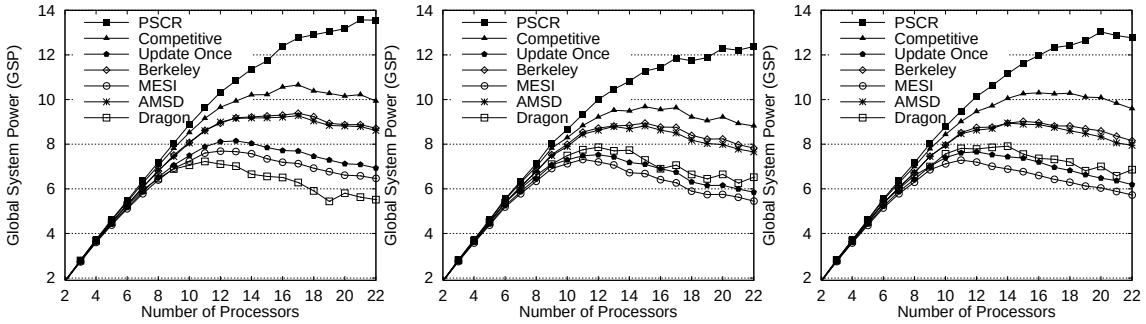

Figure 6.11. High-latency case study (64-bit bus, 128-byte block size): Global System Power. In this case, the increased latency limits the system scalability.

6.4 Influence of bus width

In the case of larger bus widths (128 to 256 bits), it has been found again that the block size that yielded an optimal performance is 128 bytes for all protocols.

All protocols provide better performance and a wider linear range. This appears quite evident by the higher GSP and, consequently, by the higher critical point values (Figure 6.12). The timing values are reported in Table 6.4.

Table 6.2. Bus timings (cycles) for the 128-bit, 256-bit bus case study (128-byte block size).

Bus-Width	READ-BLOCK		WRITE	INVALIDATE	UPDATE-BLOCK
	MEMORY	CACHE			
128-bit	32	24	5	18	5
256-bit	24	16	5	10	5

The difference between PSCR and the other protocols becomes lower, since the cost of the read-block transaction decreases, whereas the cost of the other transactions keeps constant. For this reason, the protocols based on non-selective invalidation are less penalized than in the previous situations. In particular, in the case of workloads consisting of parallel applications and a 256-bit bus width, Berkeley and AMSD approach the PSCR performance. In this case, it is clear that Dragon does not take advantage of the increased bus width.

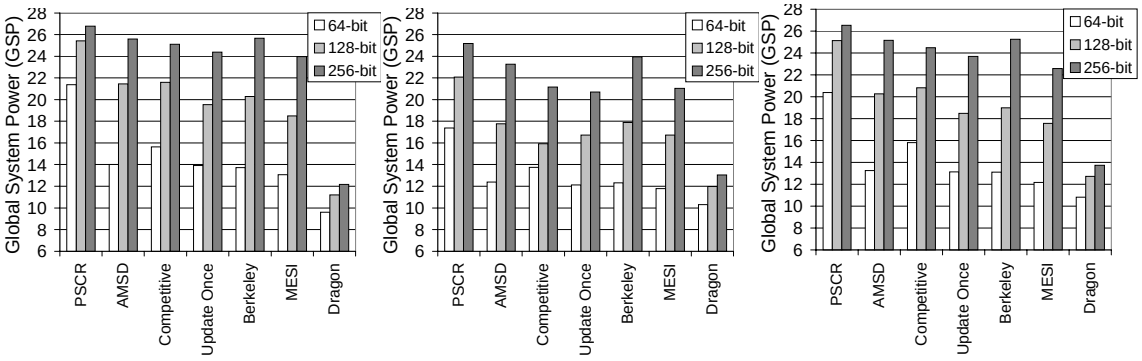

Figure 6.12. The GSP of PSCR against all other protocols, in the critical point (64-bit, 128-bit, 256-bit bus). PSCR continues to have the best performance even if the differences shrink as the bus width increases.

6.5 Influence of the Scheduling Policy

As noticed in Chapter 1, process migration allows the system to have load balancing in multiprocessor systems without requiring particular efforts to the programmer. Process migration has two negative effects of on global performance: i) a peak of misses due to the loading of the working set of the new process; and ii) the generation of passive shared copies, which happens when a migrated process is rescheduled in a short time on a different processor.

The scheduling strategies based on cache-affinity [Markatos92], [Squillante93], [Torrellas95], [Vaswani91] can reduce both effects. In fact, in this case each process is preferably rescheduled on the same processor on which it was previously executed, so that part of its working set is still resident in cache.

In this Section, the results of simulations, in which a cache-affinity scheduling strategy is used along with the PSCR protocol, are discussed. The system performance diagrams (Figure 6.13) show that, for a low number of processors, Dragon appears to obtain the highest benefits from this solution, and its performance approaches the PSCR values.

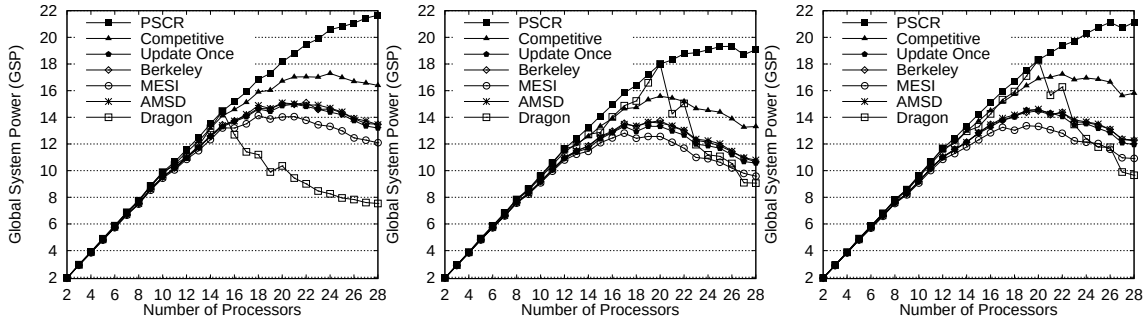

Figure 6.13. Cache-affinity case study (64-bit bus, 128-byte block size): Global System Power. PSCR eliminates passive sharing effects even when cache affinity fails.

However, the cache-affinity technique cannot provide good results in all workload conditions. In particular, it seems to be inefficient when the effects of process migration become more relevant. Indeed, while the number of cache misses due to context switch remains roughly constant, the coherence overhead induced by passive shared copies depends on the interval between the instant in which a process is suspended from execution and its subsequent resumption. The ordinary cache replacement activity progressively eliminates also possibly passive shared copies, and therefore, if a process is suspended for a long time interval, the effects of process migration on coherence overhead are drastically reduced. This time interval statistically decreases when the number of ready-to-run

processes is comparable to the number of processors (Figure 6.14). In this case, the probability that a process could be rescheduled on the same processor where it was previously executed also decreases, and this is the main reason for the failure of the cache-affinity scheduling strategy and for the drop in the Dragon performance when the processor number roughly equals half the number of processes running in the system (Figure 6.13).

In systems where cache-affinity scheduling policy is implemented, the adoption of PSCR protocol can provide relevant benefits, because it drastically reduces that passive sharing which is still present.

In addition, from Figure 6.14, more relevant passive sharing effects are expected in Const–30 workload compared to CVar50–30/MVar50–30, for a given number of processors. This is due to the smaller mean resumption distance, as already observed. Comparing the behavior of Dragon for these three workloads in Figure 6.13, the same results are expected since, in the case of Const–30, this protocol reaches its saturation point for a number of processors lower than in CVar50–30/MVar50–30. Finally, from Figure 6.13, it can be noticed again the potential of Write-Update schemes like Dragon, which can achieve better performance than Write-Invalidate, once that passive shared copies are somehow reduced.

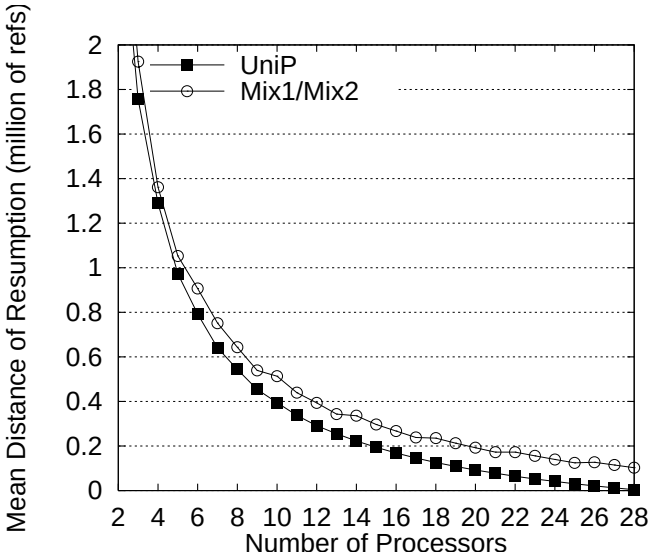

Figure 6.14. The average value of resumption distance vs. the number of processors for Const–30, CVar50–30, and MVar50–30, in the case of Cache-Affinity scheduling. The higher values for CVar50–30 and MVar50–30, compared to Const–30 workload, are due to the higher number of processes, for a given number of processors.

6.6 The Multithreaded Case

From a preliminary study [Giorgi97c] has been found that the use of multiple contexts permits to achieve a lower bus utilization if there is enough cache associativity. The longer lifetime of cached blocks produces more passive sharing and thus PSCR can take advantage from this situation.

Moreover, for a given number of processes, the increased number of contexts, produces a higher number of *virtual-processors*. This situation may be dangerous for the high level scheduler, in a way similar to what happens in the cache-affinity case. In fact, the difference between the number of processes and the number of virtual-processors may favor the production of passive copies, and thus PSCR.

The advantage of PSCR in multithreaded architectures can be explained using the following graphs. In this case, in order to get results for a 28-processor multiprocessor having 3-context multithreaded processors a new workload has been generated containing 86 independent sequential processes instead of 30. The name of this workload is Const–86, and its characteristics are essentially the same as Const–30, except for the total number of processes.

The simulations has been carried out as in the reference case, for a 64-bit bus-width, a 64-byte cache block size, a 256-KByte cache size, but for a 2-way cache. As can be observed from the first graph of Figure 6.15 the results are very close to what obtained for the Const–30 workload.

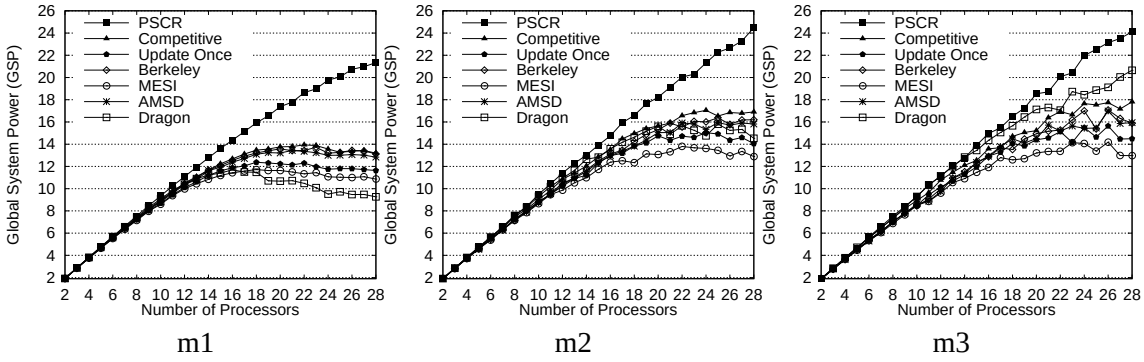

Figure 6.15. The GSP of PSCR against all other protocols, for different number of contexts per processor: m1, m2, m3 stands for 1, 2, 3 contexts per processor respectively (64-bit bus, 128-byte block, 2-way set associative, 86 process trace). PSCR outperforms all the other protocols.

As the number of contexts per processor is increased, however, a lower bus utilization and a higher Global System Power is achieved (Figures 6.16 and 6.15, respectively)

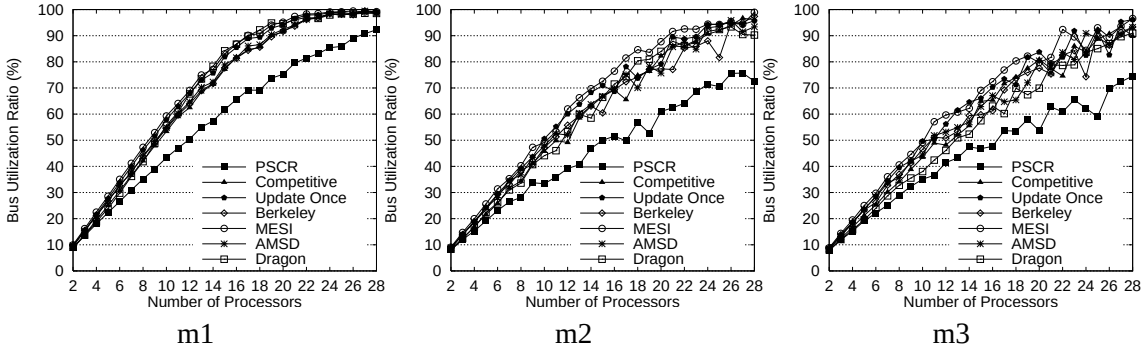

Figure 6.16. The BUR of PSCR against all other protocols, for different number of contexts per processor: m1, m2, m3 stands for 1, 2, 3 contexts per processor respectively (64-bit bus, 128-byte block, 2-way set associative, 86 process trace). PSCR outperforms all the other protocols.

The effectiveness of PSCR as the number of contexts is increased can be better observed in the Processor/Bus Efficiency graphs. PSCR is a very promising solution for this kind of architecture.

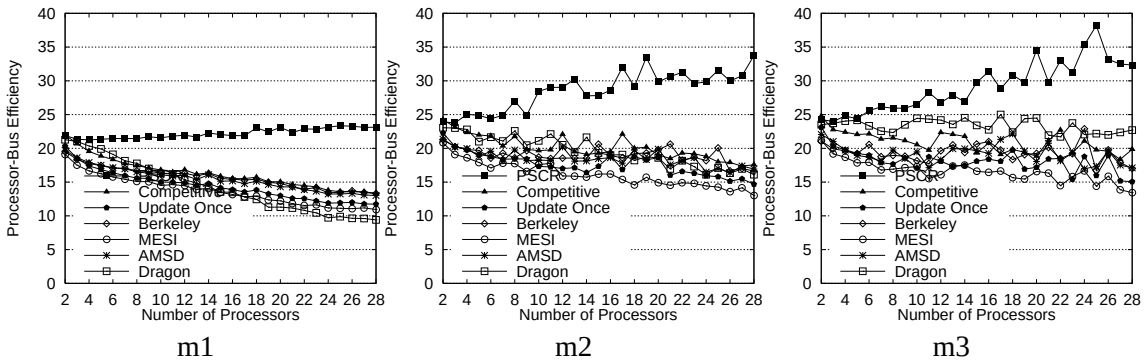

Figure 6.17. The PBE of PSCR against all other protocols, for different number of contexts per processor: m1, m2, m3 stands for 1, 2, 3 contexts per processor respectively (64-bit bus, 128-byte block, 2-way set associative, 86 process trace). PSCR outperforms all the other protocols.

6.7 Enhancing the PSCR Performance: PSCR+

As noticed in Chapter 2, the write-run statistics of a parallel application can be used to decide the threshold for switching from WU to a WI policy. In this case study, the focus is on how to improve the performance of PSCR, compared with its basic version, by using a per-block WU or WI policy based on the mean WRL (Write-Run Length) dynamically experienced by that block.

In this enhanced version of PSCR (named PSCR+), the protocol is supposed to treats those S-blocks that dynamically have a WRL higher than a given (possibly statically detected) threshold value WRLTH as they were P-blocks. Table 6.3 shows the new values of GSP obtained in the case of CVar50–30 workload and 21 processors, when WRLTH assumes different values.

Table 6.3. GSP for the CVar50–30 workload and 21 processors when the S-blocks are treated as P-blocks as they exhibit a write-run higher than a given thresholdWRL_{TH}.

WRL_{TH}	**1**	**2**	**3**	**4**	**5**	**REFERENCE CASE STUDY**
GSP	1577	1681	1750	1674	1641	1633

Chapter 7

Complexity

In the following, the implementation cost and complexity of PSCR is estimated against the other protocols based on three parameters: i) number of logical states (i.e. bits in the status field of each block), ii) type of bus transaction, and iii) number of required bus signals. Table 7.1 summarizes the features of PSCR and the other protocols used for comparison. PSCR also needs additional hardware and software support that is described in detail below. Details regarding a specific bus implementation are not considered here.

Table 7.1. Complexity of different coherence protocols.

Protocol	**PSCR**	**Dragon**	**Berkeley**	**Competitive**
Number of states	5	4	4	5+5 (5+20)
Number of bits	3	2	2	4 (5)
Type of bus transactions	RB, W, UP	RB, W, UP	RB, I, UP	RB, W, UP
Type of bus signals	DI, L_1, L_2	DI, SH	DI, RS/RO	DI, SH, EII, EIO

Protocol	**Update-Once**	**MESI**	**AMSD**
Number of states	6	4	7
Number of bits	3	2	3
Type of bus transactions	RB, W, UP	RB, W, UP	RB, I, UP
Type of bus signals	DI, SH	DI, RWITM	DI, SH, M

7.1 Number of Logical States

For all the coherence protocols, the original papers about them have been considered (see Chapter 2). In particular, for the Competitive Snooping protocol, each block has an associated counter, which is decremented whenever a write transaction involving the block is observed, in order to invalidate the block when the maximum allowed number of write transactions is reached. The size of such counter depends on the ratio between read-block transaction and write transaction time costs.

Table 7.1 presents the minimum and maximum values for the configurations examined in the present paper. The number of logical states is the sum of three components: states of the counter, states of the basic WU scheme (in this case the Dragon protocol), plus the invalid state.

7.2 Bus Transactions and Signals

PSCR, MESI, Dragon, Update-Once and Competitive Snooping make use of three different kinds of bus transaction: read-block (to fetch a missed block), write (to update multiple cached copies), and update-block (to write back dirty copies when they need to be destroyed for replacement).

Berkeley does not adopt the write transaction, but needs an invalidation transaction to destroy remote copies in the case of write operation on a shared copy. Two different kinds of bus operation are employed to fetch a missed block: read-shared (analogous to the Dragon read-block) on read miss, and read-for-ownership, which invalidates all remote copies, in the case of write miss. A proper bus signal (RS/RO) makes the distinction between the two kinds of transaction.

AMSD uses both an explicit invalidate transaction as in Berkeley, and a local invalidation mechanism of blocks deemed as migratory.

In the case of MESI protocol, a special situation happens when a miss involves a modified remote copy. Since the shared state is not split in shared-clean and shared-dirty, the memory should be updated before the transition to shared state of these copies. Cache designers have adopted different solutions to manage this situation. In the present work this solution is implemented: on a read miss, the remote cache aborts the bus transaction, then it writes back the copy to the main memory, and allows the originating cache to retry the bus transaction. Both copies will become shared. On a write miss, the cache starts a ”Read With Intent To Modify” bus operation by means of RWITM line. The remote cache writes back the copy to the memory and invalidates the local copy. The originating cache saves the copy in Modified state and performs the write locally.

PSCR adopts the same bus transactions as Dragon, and it also needs a bus signal (L_1) to allow slave cache to have a different behavior during a read-block transaction.

As for the bus signals, all protocols need at least the Data Intervention (DI) bus signal. The DI signal is used by a slave cache to substitute for main memory during a read-block transaction when it holds a dirty copy (to guarantee coherence) or a private clean copy (to improve efficiency). When this signal is raised by a remote cache, the cache read-block transaction is used, which is usually faster because of the lower cache latency compared with memory latency.

In the present evaluation MESI protocol is supposed to the DI line to abort a transaction, and the RWITM line to signal the ”Read With Intent To Modify” Read-Block transaction.

Dragon, Competitive Snooping, Update-Once, AMSD and PSCR use the SH signal, during a bus

transaction, to notify that a cache holds a copy of the pertinent block, which therefore has to be considered shared. As seen above, in PSCR protocol this function is accomplished by the L_2 line, which is also used to signal data transfer of dirty P-blocks fetched by a remote processor.

The migratory detection mechanism of AMSD needs to signal that the copy is migratory during the bus transaction used to respond to read misses, write misses, and invalidation requests. An additional line (M) is necessary for signaling the migratory condition.

Finally, Competitive Snooping needs two signals (EII, Enable-to-Invalidate-In, and EIO, Enable-to-Invalidate-Out), used to implement the distributed arbitration scheme to elect a cache in which to decrement the counter of write transactions [Karlin86].

As shown in Table 7.1, the complexity of PSCR is comparable to that of the other coherence protocols examined in the present paper. In fact, the number of logical states is one more than MESI, Dragon, and Berkeley, and fewer states than AMSD, Competitive Snooping, and Update-Once. PSCR has the same number of bus transactions of all other examined protocols. Finally, it has the same number of additional bus signals as AMSD, one more than Dragon, Berkeley, MESI, and Update-Once, and one less than Competitive Snooping.

7.3 Additional Hardware and Software Support for PSCR

As introduced in Section 3.2 the used approach is both hardware and software based. In the simplest implementation, the extra hardware consists of an extra bit (P-bit) for each page descriptor in TLB and a signal of the processor-cache bus. As for the software, the compiler is supposed to be able to organize data in such a way that the kernel or the run time support could mark the pages containing private data. This is normally accomplished by compiler and kernel in order to manage the virtual memory. Thus, the only additional support required is the extra wire between processor and cache. In particular, no additional complexity is required to manage dynamically allocated memory, since the allocation function usually needs to specify if the data should be private or shared. In the case of multithreading programming environments, in which all data are placed in a shared space, an additional effort from the compiler could be helpful. First, data allocated into the private stack are easily detectable. The situation changes for private data allocated in the shared space, or when shared data exhibit long write-runs. In this case, as observed in Section 6.7, marking those data as private data would improve the global performance. This can be done by means of a compiler tool that could detect which variables might profitably be treated as private [Xia96b]. The technique can be profiling based, data-flow based [Aho86], or relying on static analysis as successfully used by Stenström et al. [Skeppstedt94], and Mowry at al. [Mowry98].

7.4 Low Level Optimizations of PSCR

In this presentation of PSCR, it has been supposed that some low-level optimizations could be integrated with the basic hardware. In particular, in case of write transactions operating on SD copy, memory updating can be avoided (Figure 3.1). A bus signal can be used to disable memory during this operation, for example the previously described DI signal. This optimization can be useful in a couple of cases: to reduce the number of transactions involving the shared memory, or in implementations that do not employ a memory write buffer. It can be noticed that L_1 and L_2 signals are used in distinct temporal intervals. L_1 specifies the type of memory block, slave units answer by means of L_2. Thus, L_1 and L_2 can be implemented by using a single multiplexed line.

Chapter 8

Conclusions

The behavior of the PSCR protocol as a function of cache organization, memory latency, bus width, scheduling policy, and number of thread-contexts has been analyzed.

The proposed protocol has been compared against six solutions based on completely different handling policies concerning shared copies: exclusive write-update (Dragon), exclusive write-invalidate (Berkeley and MESI), or dynamically switching between the two (Update-Once, Competitive Snooping) or using an adaptive detection of migratory copies (AMSD). It has been showed that the proposed protocol represents a good solution for a shared-memory shared-bus multiprocessor in all the cases under consideration.

The improvement is particularly useful in many different cases: i) when read cost is higher than write cost, as is the case for high memory latency, in current systems; ii) when the block size is increased; iii) when the number of thread-contexts is increased.

Multiprocessors represent a significant percentage of recent architectural solutions for workstations. These machines are rarely used to speed-up a single parallel application, rather they are employed as servers to achieve a higher throughput by running multiple processes simultaneously. These machines typically use an Unix-like multitasking operating system. In the next future multithreaded architectures will be adopted along with support for multiple processors on a single chip.

Based on these considerations, three distinct workload models to evaluate the performance of the proposed protocol have been used: the first workload only includes Unix commands and single-process applications, whereas the others also include parallel applications with different sharing pattern (coarse/medium grain).

In the target system of this analysis, load balancing is obtained by allowing process migration, which distributes the computation workload among the available processing units. This migration determines the generation of passive shared copies, which induce a relevant amount of coherence

overhead. It has been showed that this protocol eliminates this overhead by operating directly during the read-block transaction consequent to a miss condition and without any significant effect on the hardware complexity. On the other hand, WI protocols do not succeed in eliminating this overhead. Once passive sharing is eliminated, WU protocols continue to be a viable strategy compared to WI ones. The selective invalidation mechanism adequately combined with its basic write-update mechanism allows PSCR to gain the benefits of an updating in bus-based architectures. I am not aware of other approaches that explicitly eliminate the overhead due to private data accesses.

The proposed solution can also be successfully employed in systems including a cache-affinity scheduler, which, as many authors have stated, not always succeeds in eliminating process migration and its effects in all workload conditions. PSCR performance can be further enhanced by using compilation techniques that recognize long write runs on shared data.

Acknowledgments

First of all, I wish to thank Professor Antonio Prete, who has been my principal advisor during the three-year period of my Research Doctorate. His suggestions and experience helped improve very considerably my way to attach problems in order to go directly to the their core, and my way to present research results in order to be effective. I thank Professor Dan Siewiorek of Carnegie Mellon University for the initial material used to validate the simulation environment. I'm particularly grateful to Gianpaolo Prina and Luigi Ricciardi for a contribution to the multiprocessor simulation design, the useful discussions about the strategy to limit passive sharing problem and preliminary evaluation of the coherence protocol. My colleague and friend Pierfrancesco Foglia contributed significantly to the validation of the performance evaluation methodology, and to the review of the past multithreaded architectures. Thanks to Steve Herrod at Stanford University for providing and helping with TangoLite. The discussions with Professor Veljko Milutinović and Professor Per Stenström put me into the right lane in order to present myself to the international scientific community. I wish to thank Professor Ali Hurson, who has been very important in encouraging me in keeping on my research.

Bibliography

[Adve91] S. V. Adve, M. D. Hill, and M. Vernon, "Comparison of Hardware and Software Cache Coherence Schemes," *Proc. of the 18th Int'l Symp. on Computer Architecture*, pp. 298–308, May 1991.

[Adve96] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, Dec. 1996.

[Agarwal90] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz, "April: a processor architecture for multiprocessing," *Proc. 17th. Int'l Symp. Computer Architecture*, pp. 104–114, May 1990.

[Agarwal91] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung, "The MIT Alawife Machine: a Large-Shared Distributed-Memory Multiprocessor," Mit/lcs/tm-454.b, MIT Laboratory for Computer Science, June 1991.

[Agarwal92] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 5, pp. 525–539, Sept. 1992.

[Agarwal93] A. Agarwal, J. Kubiatowicz, D. Kranz, B. H. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: an evolutionary processor design for large-scale multiprocessors," *IEEE Micro*, vol. 13, no. 3, pp. 48–61, June 1993.

[Agarwal95] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-J. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," *Proc. 22th Int'l Symp. on Computer Architecture*, pp. 2–13, 1995.

[Aho86] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Alverson90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *1990 Int'l Conf. on Supercomputing*, June 11-15 1990.

[Alverson92] G. A. Alverson, R. Alverson, D. Callahan, and B. Koblenz, "Exploiting Heterogeneous Parallelism on a Multi-threaded Multiprocessor," *Conf. proceedings / 1992 Int'l Conf. on Supercomputing, July 19–23, 1992, Washington, DC*, Int'l Conf. ON SUPERCOMPUTING 1992; 6th, New York, NY 10036, USA, pp. 188–197, 1992.

[Anderson94] C. Anderson and J.-L. Baer, "Design and Evaluation of a Subblock Cache Coherence Protocol for Bus-Based Multiprocessors," Tech. Rep. TR-94-05-02, Department of Computer Science and Engineering, University of Washington, May 1994.

[Anderson96] C. Anderson and A. R. Karlin, "Two Adaptive Hybrid Cache Coherency Protocols," *Proc. Second Int'l Symp. on High-Performance Computer Architecture*, pp. 303–313, Feb. 1996.

[Archibald86] J. K. Archibald and J. L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Comp. Systems*, vol. 4, pp. 273–298, Apr. 1986.

[Archibald87] J. K. Archibald, *The Cache Coherence Problem in Shared-Memory Multiprocessor*, PhD thesis, University of Washington, Mar. 1987.

[Archibald88] J. K. Archibald, "A Cache Coherence Approach For Large Multiprocessor Systems," *Proc. Int'l Conf. on Supercomputing*, pp. 337–345, July 1988.

[Berg95] S. Berg, M. Philipose, F. Pighin, and R. Schimkat, "Contrasting Hardware and Software Solutions to False Sharing on Bus-Based Shared Memory Multiprocessors," Tech. Rep., Department of Computer Science and Engineering, University of Washington, Mar. 1995.

[Bitar86] P. Bitar and A. M. Despain, "Multiprocessor Cache Synchronization — Issues, Innovations, Evolution," *Proc. 13th Int'l Symp. on Computer Architecture*, Tokyo, Japan, pp. 424–433, June 1986.

[Boothe93] R. F. Boothe, *Evaluation of Multithreading and Caching in Large Shared Memory Parallel Computers*, Also tech report ucb-csd-93-766, University of California, Berkeley - Department of Computer Science, July 1993.

[Brorsson94] M. Brorsson and P. Stenström, "Modeling Accesses to Migratory and Producer-Consumer Characterized Data in a Shared Memory Multiprocessor," *Proc. 6th Symp. on Parallel and Distributed Processing*, pp. 612–619, Oct. 1994.

[Carter95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Techniques for reducing consistency-related communication in distributed shared-memory systems," *ACM Trans. Computer Systems*, pp. 205–243, 8 1995.

[Censier78] L. M. Censier and P. Feautrier, "A New Solution to the Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, vol. 27, no. 12, pp. 1112–1118, Dec. 1978.

[Chen93] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The effect of code expanding optimizations on instruction cache design," *IEEE Trans. Computers*, vol. 42, no. 9, pp. 1045–1057, Sept. 1993.

[Chen96] Y. Y. Chen, J. K. Peir, and C. T. King, "Performance of shared cache on multithreaded architectures," *Proc. Fourth Euromicro Workshop on Parallel and Distributed Processing - PDP '96. Braga, Portugal*, pp. 541–8, Jan. 1996.

[Cheong88] H. Cheong and A. V. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation," *Proc. 15th Int'l Symp. on Computer Architecture*, Honolulu, Hawaii, pp. 299–307, May–June 1988.

[Cheriton91] D. R. Cheriton and H. A. Goosen, "Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture," *IEEE Computer*, vol. 24, no. 2, pp. 33–64, Feb. 1991.

[Cox93] A. L. Cox and R. J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proc. 20th Int'l Symp. on Computer Architecture*, San Diego, California, pp. 98–108, May 1993.

[Culler91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 164–175, Apr. 1991.

[Culler98] D. Culler, J. O. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1998.

[Dec93] *DECChip 21064 - A RISC Microprocessor Preliminary Data Sheet*, DEC - Digital Equipment Corporation, Maynard, MA, 1993.

[Diep95] T. A. Diep, *A Visualization-based Microarchitecture Workbench*, PhD thesis, Carnegie Mellon University, Aug. 1995.

[Dubois88] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, vol. 21, no. 2, pp. 9–22, Feb. 1988.

[Dubois91] M. Dubois and J.-C. Wang, "Shared Block Contention in a Cache Coherence Protocol," *IEEE Trans. Computers*, vol. 40, no. 5, pp. 640–644, May 1991.

[Dubois91b] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen, "Delayed Consistency and its Effect on the Miss Rate of Parallel Programs," *Proc. 4th Conf. on Supercomputing*, Alburquerque, NM, USA, pp. 197–207, Nov. 1991.

[Dubois93] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström, "The Detection and Elimination of Useless Misses in Multiprocessors," *20th Int'l Symp. on Computer Architecture*, May 1993.

[Eggers88] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation," *Proc. 15th Int'l Symp. on Computer Architecture*, Honolulu, Hawaii, pp. 373–382, May 1988.

[Eggers89] S. J. Eggers, "Simulation Analysis of Data Sharing in Shared Memory Multiprocessors," Phd thesis ucb/csd 89/501, Univ. of California, Berkeley, Apr. 1989.

[Eggers89b] S. J. Eggers and R. H. Katz, "Evaluating the Performance of Four Snooping Cache Coherency Protocols," *Proc. 16th Int'l Symp. on Computer Architecture*, Jerusalem, Israel, pp. 2–15, May 1989.

[Eggers90] S. J. Eggers, D. Keppel, E. J. Koldinger, and H. M. Levy, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," *Proc. 1990 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 37–47, May 1990.

[Eggers91] S. J. Eggers, "Simplicity versus Accuracy in a Model of Cache Coherency Overhead," *IEEE Trans. Computers*, vol. 40, no. 8, pp. 893–906, Aug. 1991.

[Eggers97] S. J. Eggers, J. Emer, H. M. Levy, J. L. Lo, R. Stamm, and D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," Tech. Rep., TR-97-04-02, 4 1997.

[Fiske95] J. A. S. Fiske, "Thread Scheduling Mechanisms for Multiple-Context Parallel Processors," Tech. Rep. AITR-1545, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts, June 1995.

[Flynn95] M. J. Flynn, *Computer Architecture, Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, 1995.

[Frank84] S. J. Frank, "Tightly coupled multiprocessor system speeds memory access times," *Electronics*, vol. 57, no. 1, pp. 164–169, Jan. 1984.

[Gee93] J. G. Gee and A. J. Smith, "Absolute and Comparative Performance of Cache Consistency Algorithms," Tech. Rep. UCB//CSD-93-753, EECS Computer Science Division, University of California, Berkeley, 1993.

[Gharachorloo90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and event ordering in scalable shared-memory multiprocessors," *Proceedings 17th Int'l Symp. on Computer Architecture*, pp. 15–26, May 1990.

[Giorgi96] R. Giorgi, C. Prete, G. Prina, and L. Ricciardi, "A Hybrid Approach to Trace Generation for Performance Evaluation of Shared-Bus Multiprocessors," *Proc. 22nd EuroMicro Int'l. Conf.*, Prague, pp. 207–214, Sept. 1996.

[Giorgi97c] R. Giorgi, P. Foglia, and C. Prete, "Bus Utilization Analysis of Multithreaded Shared-Bus Multiprocessors," *Proc. of 9th Int'l Conf. on Parallel and Distributed Computing and Systems*, Washington, D.C., pp. 24–29, Oct. 1997.

[Giorgi97e] R. Giorgi, C. Prete, G. Prina, and L. Ricciardi, "Trace Factory: Generating Workloads for Trace-Driven Simulation of Shared-Bus Multiprocessors," *IEEE Concurrency*, vol. 5, no. 4, pp. 54–68, Oct. 1997.

[Goldschmidt93] S. R. Goldschmidt and J. L. Hennessy, "The Accuracy of Trace-Driven Simulations of Multiprocessors," *Proc. ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 146–157, May 1993.

[Goldschmidt93b] S. R. Goldschmidt, *Simulation of Multiprocesors, Speed and Accuracy*, doctoral dissertation, Stanford University, Stanford, Calif., June 1993.

[Goodman83] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Int'l Symp. on Computer Architecture*, Stockholm, Sweden, pp. 124–131, June 1983.

[Grahn96] H. Grahn and P. Stenström, "Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Data Detection," *Journal of Parallel and Distributed Computing*, vol. 39, no. 2, pp. 168–180, Dec. 1996.

[Gruenewald96] W. Gruenewald and T. Ungerer, "A multithreaded processor designed for distributed shared memory systems," *Proc. 22nd EuroMicro Int'l. Conf.*, Prague, pp. 592–599, Sept. 1996.

[Gupta91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, "Comparative Evaluation of Latency Tolerating Techniques," *Proc. 18th Int'l Symp. on Computer Architecture*, pp. 254–263, May 1991.

[Gupta92] A. Gupta and W.-D. Weber, "Cache Invalidation Patterns in Shared-Memory Multiprocessors," *IEEE Trans. Computers*, vol. 41, no. 7, pp. 794–810, July 1992.

[Halstead88] J. R. H. Halstead and T. Fujita, "MASA: A Multithreaded processor architecture for parallel symbolic computing," *Proc. of the 15th Int'l Symp. on Computer Architecture*, pp. 443–451, May 1988.

[Hennessy96] J. Hennessy and D. A. Petterson, *Computer Architecture a Quantitative Approach, 2nd edition*, Morgan Kaufmann, 1996.

[Hirata92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An elementary processor architecture with simultaneous instruction issuing from multiple threads," *Proc. 19th Int'l Symp. on Computer Architecture*, Gold Coast, Australia, pp. 136–145, May 1992.

[Hollyday92] M. A. Hollyday and C. S. Ellis, "Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulations," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 1, pp. 97–109, Jan. 1992.

[Hwang93] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993.

[Iannucci94] R. A. Iannucci, G. R. Gao, R. H. Halstead, and B. J. Smith, eds., *Multithreaded computer architecture: a summary of the state of the art*, Kluwer Academic, 1994.

[Intel91] *i860 XP Microprocessor Data Book*, Intel Corporation, Santa Clara, CA, 1991.

[Jeremiassen95] T. E. Jeremiassen and S. J. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations," *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 179–188, Aug. 1995.

[Kadiyala95] M. Kadiyala and L. N. Bhuyan, "A Dynamic Cache Sub-block Design to Reduce False Sharing," Tech. Rep. TR95-010, Texas A&M University, 1995.

[Karlin86] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator, "Competitive Snoopy Caching," *Proc. 27th Symp. on Foundations of Computer Science*, pp. 244–254, Oct. 1986.

[Katz85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," *Proc. 12th Int'l Symp. on Computer Architecture*, pp. 276–283, June 1985.

[Lebeck95b] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," *Proc. 22nd Int'l Symp. on Computer Architecture*, New York, pp. 48–59, June 22–24 1995.

[Lee90] J. Lee and U. Ramachandran, "Synchronization with Multiprocessor Caches," *Proc. 17th Int'l Symp. on Computer Architecture*, Seattle, WA, pp. 27–39, June 1990.

[Lee94] G. Lee, B. Quattlebaum, and L. Kinney, "Protocol Mapping for a Bus-Based COMA Multiprocessor," Tech. Rep. DICE #4, Dept. of Electrical Engineering, University of Minnesota, Mar. 1994.

[Lo97] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen, "Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading," *Trans. Computer Systems*, 8 1997.

[Magnusson95] P. S. Magnusson and B. Werner, "Efficient Memory Simulation in SimICS," *28th Simulation Symp.*, Phoenix, April 1995.

[Mankovic87] T. E. Mankovic, V. Popescu, and H. Sullivan, "CHoPP principles of operations," *Proc. of 2nd Int'l Supercomputer Conf.*, pp. 2–10, May 1987.

[Markatos92] E. P. Markatos and T. J. LeBlanc, "Load Balancing vs. Locality Management In Shared-Memory Multiprocessors," *Proc. 1992 Int'l Conf. on Parallel Processing. Volume 1: Architecture*, Ann Arbor, MI, pp. 258–267, Aug. 1992.

[Matsumoto93] T. Matsumoto and K. Hiraki, "Dynamic Switching of Coherent Cache Protocols and Its Effects on Doacross Loops," *1993 ACM Int'l Conf. on Supercomputing*, Tokyo, pp. 328–337, July 1993.

[McCreight84] E. M. McCreight, "The Dragon computer system: an early overview," *NATO Advanced Study Institute on Microarchitecture of VLSI Computer*, Urbino, Italy, July 1984.

[Miksch96] A. Miksch and W. Damm, "MSparc: A multithreaded Sparc," *Lecture Notes in Computer Science*, vol. 1123, pp. 461–469, 1996.

[Miller95] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadan, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool," *IEEE Computer*, pp. 37–46, Nov. 1995.

[Milutinovic98] V. Milutinović, *Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lesson Learned*, IEEE Computer Society Press, los Alamitos, California, USA, (in preparation), http://galeb.etf.bg.ac.yu/ vm/books.html/newbookalpha.zip, 1998.

[Mips92] *MIPS R4000 Microprocessor User's Manual*, Mips Technologies, Mountain View, CA, 1992.

[Mogul91] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," *Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, USA, pp. 75–84, Apr. 1991.

[Mowry98] T. C. Mowry, C. Chan, and A. Lo, "Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory," *Fourth Int'l Symp. on High-Performance Computer Architecture (HPCA-4)*, Feb. 1998.

[Muller96] H. L. Muller, P. W. A. Stallard, and D. H. D. Warren, "Multitasking and Multithreading on a Multiprocessor with Virtual Shared Memory," *Proc. 2nd Int'l Symp. on High-Performance Computer Architecture*, San Jose, California, pp. 212–221, Feb. 1996.

[Nilsson94] H. Nilsson and P. Stenström, "An Adaptive Update-Based Cache Coherence Protocol for Reduction of Miss Rate and Traffic," *Parallel Architectures and Languages Europe*, pp. 363–374, July 1994.

[Noakes93] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-machine multicomputer: An architectural evaluation," *Proceedings of the 20th Int'l Symp. on Computer Architecture*, pp. 224–235, May 1993.

[Pancake95] C. M. Pancake, M. L. Simmons, and J. C. Yan, "Performance Evaluation Tools for Parallel and Distributed Systems," *IEEE Computer*, pp. 16–19, Nov. 1995.

[Pancake95b] C. M. Pancake, M. L. Simmons, and J. C. Yan, "Performance Evaluation Tools for Parallel and Distributed Systems," *IEEE Parallel and Distributed Technology*, pp. 14–20, Winter 1995.

[Papadopulos91] G. M. Papadopulos and K. R. Traub, "Multithreading: A revisionist view of dataflow architectures," *Proc. of the 18th Int'l Symp. on Computer Architecture*, pp. 342–351, May 1991.

[Papamarcos84] M. Papamarcos and J. Patel, "A low overhead coherence solution for multiprocessors with private cache memories," *Proc. 11th Int. Symp. Computer Architecture*, pp. 348–354, June 1984.

[Perl96] S. E. Perl and R. L. Sites, "Studies of Windows NT performance using dynamic execution traces," *Operating System Review*, vol. 30, pp. 169–83, 1996.

[Prete90] C. A. Prete, "A new solution of coherence protocol for tightly coupled multiprocessor systems," *Microprocessing and Microprogramming*, vol. 30, no. 1-5, pp. 207–214, 1990.

[Prete91] C. A. Prete, "RST Cache Memory Design for a Tightly Coupled Multiprocessor System," *IEEE Micro*, vol. 11, no. 2, pp. 16–19, 40–52, Apr. 1991.

[Prete95] C. A. Prete, G. Prina, and L. Ricciardi, "A Trace-Driven Simulator for Performance Evaluation of Cache-Based Multiprocessor System," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 9, pp. 915–929, Sept. 1995.

[Prete95b] C. A. Prete, G. Prina, and L. Ricciardi, "A Selective Invalidation Strategy for Cache Coherence," *IEICE Trans. Information and Systems*, vol. E78-D, no. 10, pp. 1316–1320, Oct. 1995.

[Prete97] C. A. Prete, G. Prina, R. Giorgi, and L. Ricciardi, "Some Considerations About Passive Sharing in Shared-Memory Multiprocessors," *IEEE TCCA Newsletter*, pp. 34–40, Mar. 1997.

[Ramachandran96] U. Ramachandran and J. Lee, "Cache-Based Synchronization in Shared Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 32, no. 1, pp. 11–27, Jan. 1996.

[Rosenblum95] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology*, pp. 34–43, December 1995.

[Rudolph84] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors," *Proc. of the 11th Int. Symp. on Computer Architecture*, pp. 340–347, June 1984.

[Sigmund96] U. Sigmund and T. Ungerer, "Evaluating a multithreaded superscalar microprocessor versus a multiprocessor chip," *Lecture Notes in Computer Science*, vol. 1123, pp. 797–800, 1996.

[Silc98] J. Silc, B. Robic, and T. Ungerer, "Asynchrony in parallel computing: From dataflow to multithreading," *Parallel and Distributed Computing Practices, Vol. 1, No. 1*, pp. 3–30, Mar. 1998.

[Singh92] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, vol. 20, no. 1, pp. 5–44, Mar. 1992.

[Sites88] R. L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM," *Proc. 15th Int'l Symp. on Computer Architecture*, Honolulu, Hawaii, pp. 186–195, May 1988.

[Skeppstedt94] J. Skeppstedt and P. Stenström, "Simple Compiler Algorithms to Reduce Ownership Overhead in Cache Coherence Protocols," *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, (ASPLOS-VI)*, October 1994.

[Skeppstedt95] J. Skeppstedt and P. Stenström, "A Compiler Algorithm that Reduces Read Latency in Ownership-Cache Coherence Protocols," *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, June 1995.

[Smith81] R. J. Smith, "Architecture and Application of the HEP Multiprocessor Computer System," *SPIE*, vol. 298, pp. 241–248, 1981.

[Squillante93] M. S. Squillante and D. E. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *IEEE Trans. Parallel Distributed Systems*, vol. 4, no. 2, pp. 131–143, Feb. 1993.

[Srbljic97] S. Srbljic, Z. G. Vranesic, M. Stumm, and L. Budin, "Analytical Prediction of Performance for Cache Coherence Protocols," *IEEE Trans. Computers*, vol. 46, no. 11, pp. 1155–73, Nov. 1997.

[Stenstrom90] P. Stenström, "A Survey of Cache Coherence Protocols for Multiprocessors," *IEEE Computer*, vol. 23, no. 6, p. 12, June 1990.

[Stenstrom93] P. Stenström, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," *20th Int'l Symp. on Computer Architecture*, pp. 109–118, May 1993.

[Stenstrom97b] P. Stenström, E. Hagersten, D. J. Lilja, M. Martonosi, and M. Venugopal, "Trends in Shared Memory Multiprocessing," *IEEE Computer, Vol. 30, No. 12*, pp. 44–50, 12 1997.

[Stunkel91] C. B. Stunkel, B. Janssens, and W. K. Fuchs, "Address Tracing for Parallel Machines," *IEEE Computer*, vol. 24, no. 1, pp. 31–45, Jan. 1991.

[Stunkel92] C. B. Stunkel, B. Janssens, and W. K. Fuchs, “Address Tracing of Parallel Systems via TRAPEDS,” *Microprocessors and Microsystems*, vol. 16, no. 5, pp. 249–261, 1992.

[Sweazey86] P. Sweazey and A. J. Smith, “A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus,” *13th Int‘l Symp. on Computer Architecture*, pp. 414–423, June 1986.

[Takahashi96] M. Takahashi, H. Takano, E. Kaneko, and S. Suzuki, “A Shared-bus Control Mechanism and a Cache Coherency Protocol for High-Performance On-Chip Multiprocessor,” *Proc. of the Second IEEE Int‘l Symp. on High-Performance Computer Architecture*, Feb. 1996.

[Thacker88] C. Thacker, L. Stewart, and E. Satterthwaite, “Firefly: a multiprocessor workstation,” *IEEE Trans. Computers*, vol. 37, no. 8, pp. 909–920, Aug. 1988.

[Thistle88] M. Thistle and B. J. Smith, “A processor architecture for Horizon,” *Proc. of Supercomputing’88*, pp. 35–41, Nov. 1996.

[Tomasevic92] M. Tomašević and V. Milutinović, “A Simulation Study of Snoopy Cache Coherence Protocols,” *Proc. of the 25th Hawaii Int’l Conf. on System Sciences (HICSS-25)*, vol. I, pp. 427–436, Jan. 1992.

[Tomasevic93] M. Tomašević and V. Milutinović, *The cache coherence problem in shared-memory multiprocessors – Hardware solutions*, IEEE Computer Society Press, Los Alamitos, CA, Apr. 1993.

[Tomasevic94] M. Tomašević and V. Milutinović, “Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors,” *IEEE Micro*, vol. 14, no. 5 and 6, pp. 52–59 and 61–66, Oct. and Dec. 1994.

[Tomasevic96] M. Tomašević and V. Milutinović, “The word-invalidate cache coherence protocol,” *Microprocessors and Microsystems*, vol. 20, pp. 3–16, Mar. 1996.

[Torrellas90] J. Torrellas, M. S. Lam, and J. L. Hennessy, “Share Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates,” *Proc. 1990 Int‘l Conf. on Parallel Processing. Volume 2: Software*, Urbana-Champaign, IL, pp. 266–270, Aug. 1990.

[Torrellas95] J. Torrellas, A. Tucker, and A. Gupta, “Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors,” *Journal of Parallel and Distributed Computing*, vol. 24, no. 2, pp. 139–151, Feb. 1995.

[Tullsen95] D. M. Tullsen, S. Eggers, and H. M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” *Proc. 22th Int‘l Symp. on Computer Architecture*, pp. 392–403, June 1995.

[Uhlig97] R. A. Uhlig and T. N. Mudge, “Trace-Driven memory simulation: a survey,” *ACM Computing Surveys*, pp. 128–170, June 1997.

[Vashaw93] B. Vashaw, “Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors,” Tech. rep. CMUCDS-93-4, Carnegie Mellon University, Pittsburgh, PA, March 1993.

[Vaswani91] R. Vaswani and J. Zahorjan, “The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors,” *Proc. 13th ACM Symp. on Operating Systems Principles*, pp. 26–40, Oct. 1991.

[Veenstra92] J. E. Veenstra and R. J. Fowler, “A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols,” *Proc. Fifth Int‘l Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, pp. 149–160, Oct. 1992.

[Veenstra94] J. E. Veenstra and R. J. Fowler, “MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors,” *Proc., 2nd Int’l. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Durham, NC, pp. 201–207, Jan. 1994.

[Veenstra94b] J. E. Veenstra and R. J. Fowler, “The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors,” Tech. Rep. TR 490, Computer Science Department, University of Rochester, Mar. 1994.

[Vernon88] M. K. Vernon, E. D. Lazowska, and J. Zahorjan, “An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols,” *Proc. 15th Int‘l Symp. on Computer Architecture*, Honolulu, Hawaii, pp. 308–317, May–June 1988.

[Xia96b] C. Xia and J. Torrellas, “Improving the Performance of the Data Memory Hierarchy for Multiprocessor Operating Systems,” *Proc. of the Second Int‘l Symp. on High-Performance Computer Architecture*, pp. 85–94, Feb. 1996.

[Yan96] J. C. Yan and S. R. Surraki, “Analyzing parallel program performance using normalized performances indices and trace transformation techniques,” *Parallel Computing*, vol. 22, pp. 1215–37, Nov. 1996.