Some Considerations About Passive Sharing in Shared-Memory Multiprocessors

Cosimo Antonio Prete, Gianpaolo Prina, Roberto Giorgi and Luigi Ricciardi

Dipartimento di Ingegneria dell'Informazione
Facoltà di Ingegneria - Università di Pisa
Via Diotisalvi, 2 - 56100 PISA, Italy
prete@iet.unipi.it

Abstract

In a multiprocessor system, process migration guarantees load balance between processors but causes passive sharing, since private data blocks of a process can become resident in multiple caches and generate useless coherence-related overhead. We propose a selective invalidation strategy to eliminate these passive shared copies. The results of trace-driven simulation prove that our strategy can result successful in a number of situations such as the typical case of a general-purpose workstation.

1. Introduction

A major issue in the design of multiprocessor architectures is the *cache coherence* problem: when two or more processors store a copy of the same memory block in their private caches and one of them performs a write operation on a location within that block, a *coherence protocol* is required in order to guarantee that each subsequent read operation by any processor may get the up-to-date value of the modified location.

For any kind of hardware organization, when the number of nodes in the system exceeds a critical value, the processor interconnection network reaches a saturation condition, due to both cache misses and coherence-related overhead. As a consequence, a drastic drop in global performance occurs.

The frequency and pattern of accesses on shared copies is the major issue concerning this overhead. Three different sources for the generation of shared copies can be observed: i) *active sharing*, which occurs when the same cached data item is referenced by processes concurrently running on different processors; ii) *false sharing*, which occurs when several processors reference separate private data items stored in the same memory block; iii) *passive* [21] or *process-migration* [1] *sharing*, which occurs when a memory block, though belonging to a private area of a process, is replicated in more than one cache as a consequence of the migration of the owner process [6], [14], [15].

Active sharing has been widely studied and many solutions to limit the induced overhead can be found in the literature [2], [4], [20]. The consequences of false sharing become more and more evident as block size increases, since a number of unrelated objects accessed by multiple distinct processors can fall into the same cache block [1], [7], [21], [22], [23].

Process migration plays an important role in a general-purpose multiprocessor environment, since it allows the programmer to develop his applications without caring about load balance. For this reason, the reduction of the number of passive shared copies represents a key point in the design of multiprocessor architectures.

Particular scheduling strategies (*cache-affinity* [18]) have been proposed to limit process migration though preserving an acceptable degree of load balancing, but these techniques cannot be successfully applied to all workload conditions.

Some recent hardware solutions for cache coherence try to approach an "optimal" behavior by switching dynamically between *write-update* and *write-invalidate* strategies: the former should be adopted with respect to shared blocks on which short write-runs [4] are reasonably expected, whereas the latter should be utilized for blocks which are likely to generate a relevant number of passive shared copies.

In the Competitive Snoopy Caching technique [11],

the switching point from *write-update* to *write-invalidate* for each cached block occurs when the number of cycles for the write broadcasts issued equals the sum of the cycles potentially needed by all processors to reread the block, had it been invalidated. This technique limits the coherence-related overhead to twice the optimal value, but at a relevant cost in hardware complexity.

Since studies on program behavior showed that in most cases data is written by one processor either many times in succession or only once, Smith [7] proposed the Update-Once protocol, which allows only one extra-update by a remote processor before invalidation. A solution which can dynamically detect migratory sharing has been proposed by Cox and Fowler [3] and by Stenström et al. [19].

In general-purpose architectures, a number of critical workload conditions exist where passive sharing turns out to be the prevalent one; a cache coherence scheme which can detect and destroy passive shared copies appears therefore quite desirable. A solution in this direction was proposed by the authors (PSCR – *Passive-Shared-Copies-Removal*) and can be applied to every snooping protocol to identify and destroy passive shared copies [15], [17]. The main purpose of the present work is that of characterizing the effects of passive sharing in a number of typical workload conditions for a general-purpose workstation. In particular, we show an example of application of the PSCR strategy in the case of a shared-bus multiprocessor, and discuss the efficiency of our solution as a function of workload and system features.

The remainder of the paper is organized as follows: Section 2 discusses the problem of passive sharing, focusing on its causes and possible solutions; Section 3 describes our solution and shows an example of its application; Section 4 concludes the paper.

2. Characterization of passive sharing

In the present Section we consider the use of a multiprocessor workstation, where the major concern is to speed-up the execution of a set of both uniprocess and parallel applications.

As an example, we consider the *UniP* workload, consisting of 30 uniprocess applications, and the *Mix* workload, consisting of 30 uniprocess applications and an additional load due to a parallel application which generates a number of processes equal to half the total number of processors available.

As uniprocess applications, we selected a number of typical Unix commands (`awk`, `cp`, `du`, `lex`, `rm` and `ls`) with different command line options, some 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, a number of users run some UNIX commands and ordinary applications. For this reason, the proposed workloads include two or three copies of the same program taken in different execution sections.

As parallel application, we consider a coarse-grain parallel program (`mp3d`), which comes from the SPLASH suite and simulates rarefied hypersonic flow; the trace generated is relative to the case of 10,000 molecules and 20 time steps.

Figure 1 shows the percentages of write operations on shared copies due to passive sharing and to the other kinds of sharing for the two workloads, respectively.

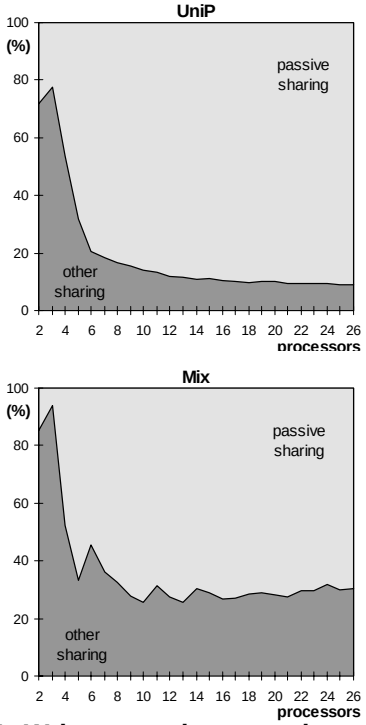

Figure 1. Write operations on shared blocks (256-Kbyte, 2-way set-associative cache, 64-byte block size, Dragon protocol, worst-case scheduling sequence)

In the case of the *UniP* workload, the accesses to shared copies are due to both kernel activity and process migration; in the *Mix* workload, the presence of the parallel application increases the amount of shared accesses due to active sharing.

A way to reduce passive sharing consists in the adoption of an invalidation-based coherence protocol; this solution can provide good performance in that it reduces the amount of the global traffic on the interprocessor connection network. Poor results, however, are

obtained when the low number of coherence-related actions due to write operations is achieved at the cost of an increased miss rate. This typically happens when a non-selective invalidation strategy is adopted, as in the case of the Berkeley protocol (Figure 2).

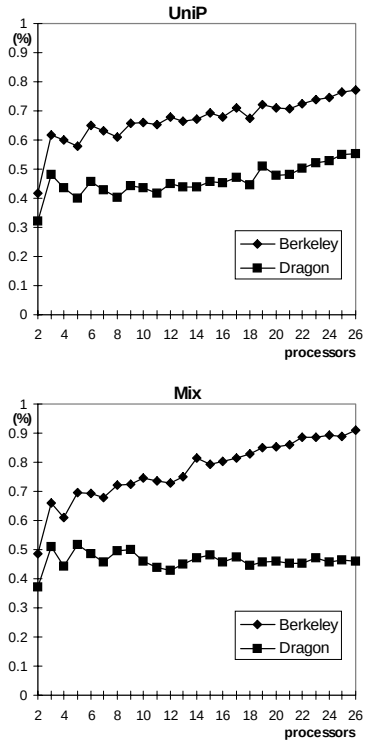

Figure 2. Miss ratio in the case of the Dragon and Berkeley protocols (256-Kbyte, 2-way set-associative cache, 64-byte block size, worst-case scheduling sequence)

Since read operations are typically synchronous for the processor, in the case of read miss the CPU must wait for the block fetching to complete before starting a new cache operation. On the other hand, write operations (in both hit/miss conditions) are 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 directly involve idle time for the CPU, even though they may significantly affect the traffic concerning miss handling and coherence-related activity. Finally, it has to be considered that the amount of data to be transferred is generally lower in the case of a write operation, even though the cost of each coherence-related operation depends on both the kind of interconnection network and the coherence strategy. The considerations just exposed justify the fact that a reduction in the number of write operations on shared copies (as a consequence of a good invalidation strategy) can result in a real advantage only if it does not involve a relevant increase in the miss rate.

Processors	P_{same}	N_{dist}	D_{sched}
12	0.72	5	500
16	0.81	3	388
20	0.03	6	310
24	0.00	7	258

Table 1. Statistics of scheduling tables in a system based on cache-affinity (UniP workload, Dragon protocol)

Two negative effects of process migration on global performance can usually be observed: i) a peak of misses needed to load the working set of the newly-scheduled process, and ii) the generation of passive shared copies when the migrated process is rescheduled in a short time on a different processor.

A solution which contributes to reduce both effects is the adoption of a scheduling strategy based on cache-affinity [18]; in this case, each process is preferably rescheduled on the same processor on which it was formerly executed, so that part of its working set is still resident in cache.

Table 1 shows some significant metrics concerning the scheduling activity for the *UniP* workload. In particular, P_{same} is the relative frequency for a process to be scheduled on the same processor where it was last executed, N_{dist} is the average number of distinct processes executed on each processor, and D_{sched} is the average distance in terms of the number of references (thousands) between two subsequent schedulings of a process.

A measure of the actual efficiency of the cache-affinity scheduling algorithm is expressed by the average number of distinct processes which alternate execution on a specific processor (N_{dist}). A lower value of such metric corresponds to higher benefits induced by the cache-affinity strategy. Intuitively, if a large number of distinct processes are scheduled on a processor in sequence, each of them will most likely find a very small portion of its working set still resident in cache in the case it is rescheduled on the same processor. As shown in Figure 3, however, the cache-affinity technique cannot succeed in eliminating passive sharing when the effects of process migration become more relevant.

Finally, a rather easy and efficient method to destroy passive shared copies consists in invalidating the copies belonging to private data areas of a process as soon as they are fetched by another processor.

This strategy ensures that a write operation in-

volving such a block will never require any coherence-related action. As soon as a private block is fetched by a remote node as a consequence of a miss condition, the only other copy possibly left by the migrated process in a remote cache is immediately invalidated. As a consequence, blocks belonging to private data areas of a process are gradually forced to "follow" the owner process in its migration, and coherence-related activity due to passive sharing is completely eliminated.

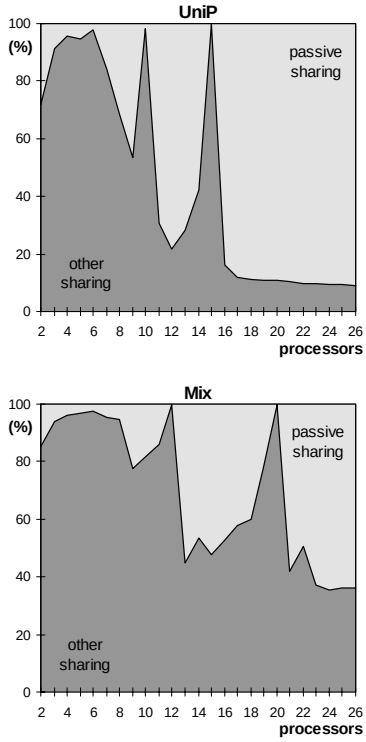

Figure 3. Write operations on shared blocks (256-Kbyte, 2-way set-associative cache, 64-byte block size, Dragon protocol, cache-affinity scheduling strategy)

3. The PSCR protocol

As an example, we will show how the technique just described has been implemented in the PSCR coherence protocol for a shared-bus multiprocessor [9].

For the sake of simplicity, we shall refer to private blocks of a process as *P-blocks*, whereas *S-blocks* are blocks belonging to a code or a shared data area.

The hardware implementation is quite simple: we assume that the processor, on each memory reference, "tells" the local cache whether the referenced address involves a private data area (P-block) of the running process or not. This information can be supplied by the MMU through each memory segment (page) descriptor, as typically happens in modern processors.

In the case of miss condition, the cache spreads this information on the shared 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 this block, the copy is immediately invalidated.

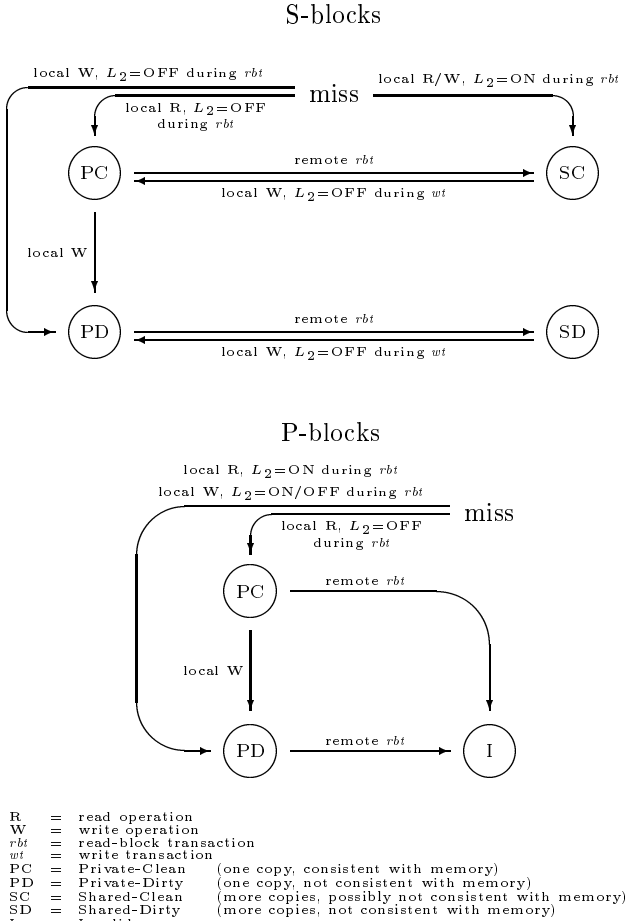

Figure 4. State transition diagrams for the PSCR protocol

A second line (L_2, handled by the "listening" caches) is required for a couple of (mutually excluding) purposes: i) during a read-block transaction involving a P-block (L_1 = ON), to indicate that a modified copy is resident in a remote cache, so that the state of the new loaded block is marked as dirty; ii) 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.

Figure 4 synthesizes the behaviour of the protocol by means of state transition diagrams for the two kinds of blocks. Regarding the hardware implementation costs, the complexity of PSCR is comparable to the one of other commonly adopted protocols based on either write-update (e.g. Dragon) or write-invalidate (e.g. Berkeley). In particular, the number of logical states is the same as in Dragon and Berkeley, and a single additional line is required compared to Dragon.

As an example of application, we consider a shared-bus multiprocessor configuration based on a 64-bit bus, with a 256-Kbyte, 2-way set-associative cache. The block size is assumed 64 bytes, since the simulations show that this value can provide the best performance for both workloads described above and for all the protocols examined. The number of processors is varied in the range 2-26; the bus timings are included in Table 2. The proposed protocol is considered together with two other protocols which exhibit a different kind of behavior concerning shared copies: Dragon [13] (based on exclusive write-update), and Berkeley [12] (exclusive write-invalidate).

Transaction	Clock cycles
memory-to-cache read-block transaction	32
cache-to-cache read-block transaction	24
write transaction	5
update-block transaction	18
invalidate transaction	5

Table 2. Bus timings

The methodology used in our analysis is trace-driven simulation. We developed a complete simulation environment consisting of a set of tools to create traces representing a predefined user workload and a particular kernel behavior, with a specific multiprocessor configuration. The environment allows the utilization of a set of *source* traces (obtained by TangoLite [5] and including only user references) to produce complete multiprocessor *target* traces. They are generated by considering the source traces, the target machine configuration (e.g. the number of processors) and the following three kernel activities: 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*. In our hybrid approach, the kernel reference stream is generated stochastically and inserted within the user reference stream. All details concerning the global environment are described in [8], [10], [16].

The simulations with the two workloads described above show that our strategy exhibits good performance in terms of both absolute GSP values (Global System Power, i.e. the sum of the average processor utilization ratios × 100) and of linearity range (Figure 5).

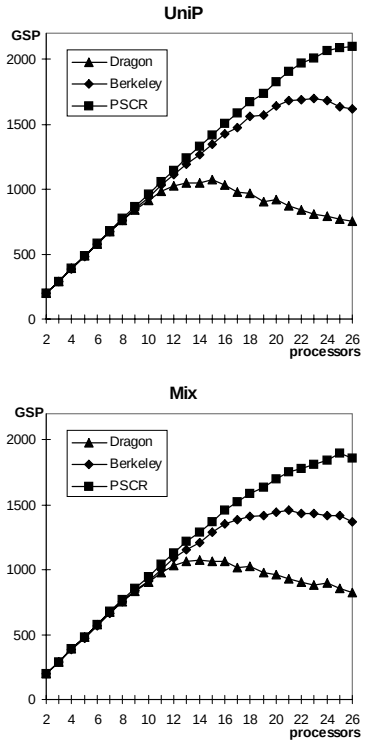

Figure 5. Global System Power for a shared-bus architecture (worst-case scheduling strategy)

The good performance of PSCR is mainly due to the low total number of bus transactions per processor, which drastically reduces the global traffic on the shared bus. Table 3 presents the number of transactions per thousand memory operations and the average bus utilization for $N = 18$ (in the case of the Berkeley protocol, the number of invalidations is considered instead of the number of write transactions).

Workload	PSCR		Dragon		Berkeley	
	UniP	*Mix*	*UniP*	*Mix*	*UniP*	*Mix*
read-block	4.1	4.3	4.6	4.4	6.7	8.2
write (inv)	4.5	13.1	45.8	43.5	2.7	4.5
bus util (%)	57.5	73.9	97.7	97.9	73.3	83.9

Table 3. Bus transactions for a 18-processor architecture

The low number of total bus transactions in the case of the PSCR protocol results from both a low miss rate and a limited number of write transactions. The two aspects are strictly related to each other, in that the

PSCR selective invalidation strategy only eliminates the useless copies of cached P-blocks, without causing further unnecessary misses. Actually, PSCR is the protocol which exhibits the lowest number of invalidations. Berkeley is the only protocol which exhibits a lower number of coherence-related bus actions due to write operations, but at the cost of an increased miss rate, due to a non-selective invalidation strategy (Figure 2).

In the second step of our analysis we consider the effects of a scheduling strategy based on cache-affinity.

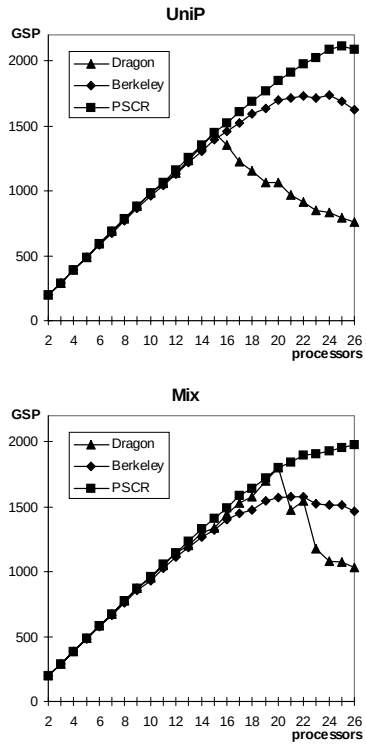

Figure 6. Global System Power for a shared-bus architecture (cache-affinity scheduling strategy)

The system performance diagrams 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. As we can see in Figure 6, however, the cache-affinity technique cannot provide good results in all workload conditions. In particular, it proves to be inefficient when the effects of process migration become more relevant. Indeed, while the number of cache misses due to context switching keeps 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 rescheduling (D_{sched} in Table 1). The ordinary cache replacement activity progressively eliminates all passive shared copies, and therefore, if the suspension time lapse of a process is large enough, the effects of process migration on coherence overhead are drastically reduced. This time lapse statistically decreases when the number of ready-to-run processes is comparable to the number of processors. In this case, the probability that a process can be rescheduled on the same processor where it was previously rescheduled also decreases (P_{same} in Table 1), 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 number of processors roughly equals half the number of processes running in the system.

In systems where a cache-affinity scheduling policy is implemented, the adoption of the PSCR protocol can therefore provide relevant benefits, in that it drastically reduces one of the effects of process migration when it cannot be eliminated by the scheduler.

4. Conclusions

Multiprocessors represent a significant percentage of recent architectural solutions for workstations; they are rarely used to speed-up individual parallel applications, and more intensively employed to achieve a high system throughput by running multiple processes simultaneously.

Process migration represents a critical issue concerning the target system of our analysis. On one hand, it can provide load balance by performing an optimal distribution of the workload among the processors; on the other hand, it generates passive shared copies, which introduce a relevant amount of coherence overhead. We proposed a selective invalidation strategy which completely eliminates this overhead by operating directly during the fetching of the block as a consequence of a miss condition (on the other hand, all write-invalidating protocols do not succeed in completely eliminating this overhead), and without affecting the hardware complexity in a significant way. The proposed solution can also be successfully employed in systems that adopt a cache-affinity scheduler, which cannot always eliminate process migration and its effects.

The performance of the PSCR protocol would be highly improved if the compiler could operate an efficient selection of all variables, so that shared data which exhibit large write-runs could be handled as private, and the choice between updating and invalidating cached copies could be performed as a function of the expected sharing pattern.

References

[1] A. Agarwal and A. Gupta, "Memory reference characteristics of multiprocessor applications under Mach", *Proc. ACM Sigmetrics*, Santa Fe, NM, pp. 215-225, May 1988.

[2] J. Archibald and J.L. Baer, "Cache coherence protocols: evaluation using a multiprocessor simulation model", *ACM Trans. Comput. Syst.*, vol. 4, n. 4, pp. 273-298, Nov. 1986.

[3] A.L. Cox and R.J. Fowler, "Adaptive cache coherency for detecting migratory shared data", *Proc. 20th Int. Symp. Comput. Arch.*, pp. 98-108, 1993.

[4] S.J. Eggers, "Simulation analysis of data sharing in shared memory multiprocessors", Ph.D. dissertation, Univ. of California, Berkeley, April 1989.

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

[6] K. Hwang, *Advanced computer architecture: parallelism, scalability, programmability*, Mc. Graw-Hill Inc., 1993.

[7] J.D. Gee and A.J. Smith, "Evaluation of cache consistency algorithm performance", *Proc. Int. MASCOTS Workshop*, San Jose, CA, Febr. 1996, pp. 236-249.

[8] R. Giorgi, C.A. Prete, G. Prina, and L. Ricciardi, "A workload generation environment for trace-driven simulation of shared-bus multiprocessors", *Proc. 30th Ann. Hawaii Int. Conf. on System Sciences*, Jan. 1997, pp. 266-275.

[9] R. Giorgi, C.A. Prete, G. Prina, and L. Ricciardi, "A cache coherence protocol based on selective invalidation for shared-bus shared-memory multiprocessors", Tech. Rep. IET-INF96-5, 1996.

[10] R. Giorgi, C.A. Prete, G. Prina, and L. Ricciardi, "Trace Factory: a workload generation environment for trace-driven simulation of shared-bus multiprocessors", Tech. Rep. IET-INF96-7, 1996.

[11] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator, "Competitive Snoopy Caching", *Algorithmica*, n. 3, pp. 79-119, 1988.

[12] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon, "Implementing a cache consistency protocol", *Proc. 12th Int. Symp. Comput. Arch.*, pp. 276-283, June 1985.

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

[14] C.A. Prete, "A new solution of coherence protocol for tightly coupled multiprocessor systems", *Microprocessing and Microprogramming*, vol. 30, n. 1-5, Amsterdam, pp. 207-214, 1990.

[15] C.A. Prete, G. Prina, and L. Ricciardi, "Reducing coherence-related overhead in multiprocessor systems", *Proc. 3rd Euromicro Workshop on Par. and Distr. Processing*, Sanremo, IEEE Computer Society Press, Jan. 1995, pp. 444-451.

[16] C.A. Prete, G. Prina, and L. Ricciardi, "A trace-driven simulator for performance evaluation of cache-based multiprocessor systems", *IEEE Trans. Parall. Distr. Syst.*, vol. 6, n. 9, Sept. 1995, pp. 915-929.

[17] C.A. Prete, G. Prina, and L. Ricciardi, "A selective invalidation strategy for cache coherence", *IEICE Transactions on Information and Systems*, vol. E78-D, n. 10, Oct. 1995, pp. 1316-1320.

[18] M.S. Squillante and D.E. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling", *IEEE Trans. Parall. Distr. Syst.*, vol. 4, n. 2, pp. 131-143, Feb. 1993.

[19] P. Stenström, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing", *Proc. 20th Int. Symp. Comput. Arch.*, pp. 109-118, 1993.

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

[21] M. Tomašević, V. Milutinović, "Hardware approaches to cache coherence in shared-memory multiprocessors", *IEEE Micro*, Oct. 1994, pp. 52-59 (part 1); Dec. 1994, pp. 61-66 (part 2).

[22] M. Tomašević, V. Milutinović, "The word-invalidate cache coherence protocol", *Microprocessors and Microsystems*, vol. 20, 1996, pp. 3-16.

[23] J. Torrellas, M.S. Lam, and J.L. Hennessy, "False sharing and spatial locality in multiprocessor caches", *IEEE Trans. Comput.*, vol. 43, n. 6, June 1994, pp. 651-663.