Process Migration Effects on Memory Performance of Multiprocessor Web-Servers

Pierfrancesco Foglia, Roberto Giorgi, Cosimo Antonio Prete

Dipartimento di Ingegneria della Informazione
Universita' di Pisa
Via Diotisalvi, 2 – 56100 Pisa, Italy
{foglia,giorgi,prete}@iet.unipi.it

Abstract. In this work we put into evidence how the memory performance of a Web-Server machine may depend on the sharing induced by process migration. We considered a shared-bus shared-memory multiprocessor as the simplest multiprocessor architecture to be used for accelerating Web-based and commercial applications. It is well known that, in this kind of system, the bus is the critical element that may limit the scalability of the machine. Nevertheless, many factors that influence bus utilization, when process migration is permitted, have not been thoroughly investigated yet. We analyze a basic four-processor and a high-performance sixteen-processor machine. We show that, even in the four-processor case, the overhead induced by the sharing of private data as a consequence of process migration, namely passive sharing, cannot be neglected. Then, we consider the sixteen-processor case, where the effects on performance are more massive. The results show that even though the performance may take advantage of larger caches or from cache affinity scheduling, there is still a great amount of passive sharing, besides false sharing and active sharing. In order to limit false sharing overhead, we can adopt an accurate design of kernel data structures. Passive sharing can be reduced, or even eliminated, by using appropriate coherence protocols. The evaluation of two of such protocols (AMSD and PSCR) shows that we can achieve better processor utilization compared to the MESI case.

1 Introduction

Multiprocessor systems are becoming more and more widespread due to both the cheaper cost of necessary components and the advances of technology. In their simplest implementation, multiprocessors are built by using commodity processors, shared memory, and shared bus, in order to achieve high performance at minimum cost [10]. Their relatively high diffusion made it common to find these machines running not only traditional scientific applications, but also commercial ones, as is the case for Web-Server systems [2],[3].

The design of such systems is object of an increasing interest. Nevertheless, we are not aware of papers that thoroughly investigate some factors that can heavily influence their performance, e.g. the effect of process migration, and the consequent passive-sharing [16]. If those aspects were considered correctly, different architectural choices would be dictated to tune the performance of Web-Server multiprocessors.

Web-Server performance is influenced by network features, I/O subsystem performance (to access database information or HTML pages) and by hardware and software architecture both from client and server side. In the following, we shall

consider a server based on shared-bus shared-memory multiprocessor, and in particular, we shall focus on the core architecture related problems, and the effects of process migration.

The design issues of a multiprocessor system are scalability and speedup. These goals can be achieved by using cache memories, in order to hide the memory latency, and reduce the bus traffic (the main causes that limit speed up and scalability). Multiple cache memories introduce the coherence problem and the need of a certain number of bus transactions (known as *coherence overhead*) that add to the basic bus traffic of cache-based uniprocessors. Thus, a design issue is also the minimization of such coherence overhead. The industry-standard MESI protocol may not be enough effective to minimize that overhead, in the case of Web-Servers.

The scheduling algorithm plays an essential role in operating systems for multiprocessors because it has to obtain the load balancing among processors. The consequent process migration generates passive sharing: private data blocks of a process can become resident in multiple caches and generate useless coherence-related overhead, which in turn may limit system performance [16].

In this paper, we shall analyze the memory hierarchy behavior and the bus coherence overhead of a multiprocessor Web-Server. The simulation methodology relies on trace-driven technique, by means of *Trace Factory* environment [8], [15].

The analysis starts from a reference case, and explores different architectural choices for cache and number of processors. The scheduling algorithm has also been varied, considering both a random and a cache affinity policy [18]. The results we obtained show that, in these systems, large caches and cache affinity improve the performance. Anyway, due to both false sharing and passive sharing, MESI does not allow achieving the best performance for Web-Server applications. Whilst a re-design of operating system kernel structures is suggested to reduce false sharing, passive sharing can be eliminated by using new coherence protocol schemes. We evaluated two different solutions for the coherence protocol (AMSD [19], [4] and PSCR [9]). The results show a sensible performance increase, especially for high performance architectures.

2 Coherence Overhead

In a shared-bus, shared-memory multiprocessors, speed-up and scalability are the design issues. It is well known that the shared bus is the performance bottleneck, in this kind of systems. To overcome the bus limitations, and achieving the design goals, we need to carefully design the memory subsystem. These systems usually include large cache memories that contribute in both hiding memory latency, and reducing the bus traffic [10], but they generate the need for a coherence protocol [14], [22], [23]. The protocol activity involves a certain number of bus transactions, which adds to the basic bus traffic of cache-based uniprocessors, thus limiting the system scalability.

In our evaluations, we have considered the MESI protocol. MESI is a Write-Invalidate protocol [21], and it is used in most of the actual high-performance microprocessors, like the AMD K5 and K6, the PowerPC series, the SUN UltraSparc II, the SGI R10000, the Intel Pentium, Pentium Pro, Pentium II and Merced. The remote invalidation used by MESI (*write-for-invalidate*) to obtain coherency has as a

drawback the need to reload the copy, if it is used again by the remote processor, thus generating a miss (*Invalidation Miss)*. Therefore, MESI coherence overhead (that is the transactions needed to enforce coherence) is due to *Invalidation Misses* and *write-for-invalidate* transactions.

We wish to relate that overhead with the kinds of data sharing that can be observed in the system, in order to detect the causes for the coherence overhead, and thus finding out the appropriate hardware/software solutions. Three different types of data sharing can be observed: i) *active sharing*, which occurs when the same cached data item is referenced by processes running on different processors; ii) *false sharing* [25], which occurs when several processors reference different data items belonging to the same memory block; iii) *passive* [23], [16] 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. Whilst active sharing is unavoidable, the other two forms of sharing are useless. The relevant overhead they produce can be minimized [24], [19], [13], and possibly avoided [9].

3 Workload Description

The Web-Server activity is reproduced by means of the Apache daemon [17], which handles HTTP requests, an SQL server, namely PostgreSQL [29], which handles TPC-D [26] queries, and several Unix utilities, which interface the various programs.

Table 1. Statistics of source traces for some UNIX utilities (64-byte block size and 5,000,000 references per application)

APPLICATION	Distinct blocks	Code (%)	Data Read	Data Write
AWK (BEG)	4963	76.76	14.76	8.47
AWK (MID)	3832	76.59	14.48	8.93
CP	2615	77.53	13.87	8.60
GZIP	3518	82.84	14.88	2.28
RM	1314	86.39	11.51	2.10
LS -AR	2911	80.62	13.84	5.54
LS -LTR (BEG)	2798	78.77	14.58	6.64
LS -LTR (MID)	2436	78.42	14.07	7.51

Web-Server software relies on server-client paradigm. The client usually requests HTML files or Java Applets by means of a Web browser. The requested file is provided and sent back to the client side. We used the Apache daemon [5]. We configured the daemon, so that it can spawn a minimum of 2 idle processes and a maximum of 10 idle processes. A child processes up to 100 requests before dying.

Besides this file transferring activity, Web-Servers might need to browse their internal database [3]. The workload thus includes a search engine, namely PostgreSQL DBMS [29]. We instructed our Web-Server to generate some queries, upon incoming calls, as specified by the TPC-D benchmark for Decision Support Systems (DSS). PostgreSQL consists of a front-end process that accepts SQL queries, and a backend that forks processes, which manage the queries. A description of PostgreSQL memory and synchronization management can be found in [27]. TPC-D

simulates an application for a wholesale supplier that manages, sells, and distributes a product worldwide. It includes 17 read-only queries, and 2 update queries. Most of the queries involves a high number of records, and perform different operations on database tables.

For completing our Web-Server workload, we considered some glue-processes that can be generated by shell scripts (`ls`, `awk`, `cp`, `gzip`, and `rm`). In a typical situation, various requests may be running, thus requiring the support of different system commands and ordinary applications.

Table 2. Statistics of multiprocess application source traces (Apache and TPC-D) and our target trace (SWEB), in case of 64-byte block size and 5,000,000 references per process.

Workload	Number of processes	Distinct Blocks	Code (%)	Data (%)		Shared blocks	Shared data (%)	
				Read	Write		Accesses	Write
APACHE	13	95343	75.14	18.24	6.61	666	1.52	0.49
TPC-D	5	8821	71.94	18.17	9.88	2573	2.7	0.79
SWEB	26	239848	75.49	17.12	7.39	3982	1.68	0.54

4 Performance Analysis

4.1 Methodology

The methodology used in our analysis is based on trace-driven simulation [20], [15], [28], and on the simulation of the three kernel activities that most affect performance: *system calls*, *process scheduling*, and *virtual-to-physical address translation* [7]. In the first phase, we produce a *source* trace (a sequence of user memory references, system-call positions, and synchronization events in case of multithreaded programs) for each application belonging to the workload, by means of a tracing tool. In the second phase, Trace Factory simulates the execution of complex workloads by combining multiple source traces, generating the references of system calls, and by simulating process scheduling and virtual-to-physical translation. Trace Factory furnishes the references (*target* trace) to a memory-hierarchy simulator [15], by using an on-demand policy. Indeed, Trace Factory produces a new reference whenever the simulator requests one, so that the timing behavior imposed by the memory subsystem conditions the reference production [8].

To detect sharing patterns, and evaluate the source of overhead, we have extended an existing classification algorithm [11] to the case of passive sharing, and the case of finite-size caches.

The Web-Server workload is constituted by 26 processes, 13 of which are spawned by the Apache daemon, 5 by PostgreSQL (corresponding to the first 5 queries of the TPC-D benchmark), and 8 processes are Unix utilities. Table 1 (for the uniprocess applications) and 2 (for the multiprocess ones) contain some statistics of the source traces that we used to generate the workload. To take into account that some requests may be using the same program at different times, we traced some commands in shifted execution sections: initial (beg) and middle (mid). Table 2 contains also the

statistics of the target workload that we used in our evaluation (SWEB). Data are related to 100 requests to the Web-Server, that produce 130 millions of references.

4.2 Simulation Results

As base case-study, a machine with 128-bit shared bus is considered. As for the number of processors, two configurations have been studied: a basic machine with 4 processors and a high-performance machine with 16 processors. For the scheduling policy, two solutions have been considered: *random* and *cache-affinity*; scheduler time slice is 200,000 references. Cache size has been varied between 32K and 2M. The simulated processors are MIPS-R10000-like; paging relays on 4-KByte-page size; the bus logic supports transaction splitting, and processor-consistency memory model [6]. The base case study timings for the simulator are summarized in Table 3.

Table 3. Numerical values of timing parameters for the multiprocessor simulator (times are in clock cycles) in the case of 64-byte block size.

CLASS	PARAMETER	TIMINGS
CPU	READ/WRITE CYCLE	2
BUS	WRITE FOR INVALIDATE TRANSACTION	5
	MEMORY-TO-CACHE READ-BLOCK TRANSACTION	72
	CACHE-TO-CACHE READ-BLOCK TRANSACTION	16
	UPDATE-BLOCK TRANSACTION	10

This paper is mainly concerned in evaluating the effects of process migration on the performance. The performance is affected by the cache misses (which essentially contribute to determine the mean memory access time) and the bus-traffic (which affects the miss cost). In the case of MESI protocol, bus traffic is due to the following transactions: *read-block* transactions (essentially due to the misses), *write-for-invalidate* transactions and *update* transactions. *Update* transactions are only a neglectable part of the bus-traffic and they do not influence greatly our analysis. The rest of the traffic is due to classical misses (sum of cold and replacement misses) and coherence traffic, constituted of *Invalidation Misses* and *write-for-invalidate* transactions.

Process migration affects the miss rate, since a migrating process has to regenerate its working set on a new processor, while it is destroying the one of the other processes. Moreover, process migration affects the coherence traffic, since it generates passive sharing: private data blocks of a process can become resident in multiple cache and generate useless coherence-related overhead [16]. Thus, our evaluation includes the analysis of both the miss rate and the coherence traffic.

We first analyze the miss rate (Figure 1) when cache size and number of ways are varied in the case of four processors. As expected, invalidation miss (i.e. true and false sharing miss, both due to kernel and user) increases with larger cache sizes. Nevertheless, the total miss rate decreases significantly as the cache size is increased up to 2M bytes, and in the case of more ways. For cache sizes above 512K bytes the difference between 2 and 4 ways becomes neglectable. In Figure 2, we detail the invalidation misses of Figure 1. Invalidation misses are basically due to the kernel, and in that case, false sharing is the main source of this overhead.

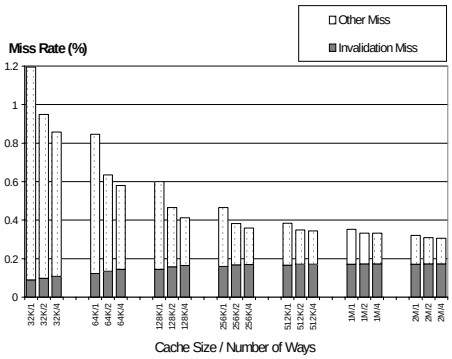

Fig. 1. Miss rate versus cache size (32K bytes, 64K bytes, 128K bytes, 256K bytes, 512K bytes, 1M bytes, 2M bytes), and number of ways (1, 2, 4). In this experiment, we had a 4-processors configuration, and random scheduling policy. "Other Miss" includes cold miss, capacity miss and replacement miss. Miss rate decreases while invalidation miss (i.e. the sum of false sharing miss and true sharing miss) increases with large cache size and more associativity.

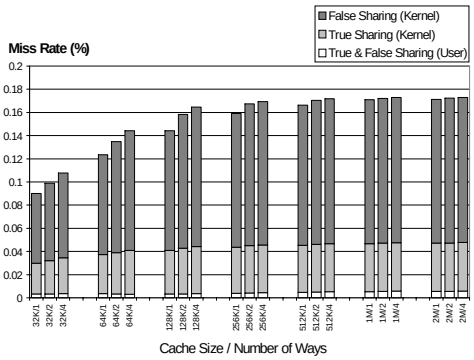

Fig. 2. Detail of the invalidation miss versus cache size (32K bytes, 64K bytes, 128K bytes, 256K bytes, 512K bytes, 1M bytes, 2M bytes), and number of ways (1, 2, 4), for a 4-processor configuration with a random scheduling policy. Invalidation misses are basically due to the kernel and false sharing is the main source of overhead.

False sharing occurs when data are used in exclusive way by different processes, and those data become physically shared. This happens when data are improperly aligned, i.e. when different data objects are placed into the same block. False sharing can be eliminated either by using special coherence protocols [24], or properly allocating the involved shared data structures [12]. This latter solution seems more suitable for our case study, since the detected false sharing is mainly due to kernel, which is a completely known part of the system at design time. Invalidation miss rate is significant for large cache sizes, whilst is neglectable for small cache sizes. Therefore, for cache sizes above 256K bytes, our results indicate that the kernel of Web-Server should be designed according to the above techniques, in order to avoid false sharing effects.

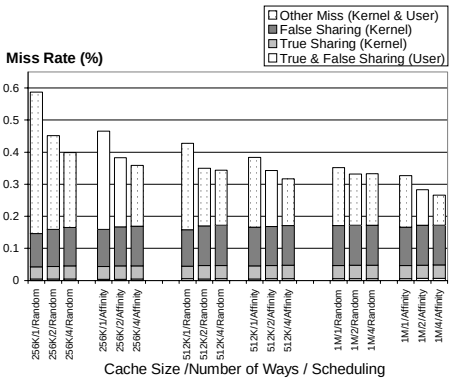

Fig. 3. Breakdown of miss rate versus cache sizes (256K bytes, 512K bytes, 1M bytes), number of ways (1, 2, 4), and scheduling policy (random, affinity), for a 4-processor configuration. Invalidation misses, and in particular, false sharing misses, increase in the 16-processor configuration.

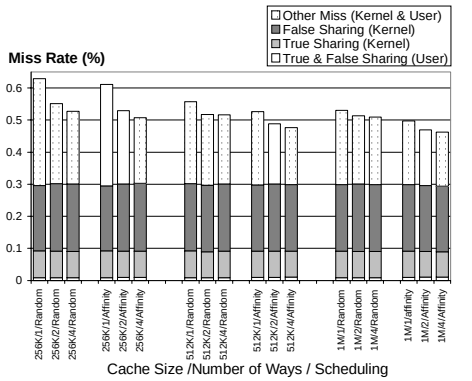

Fig. 4. Breakdown of miss rate versus cache sizes (256K bytes, 512K bytes, 1M bytes), number of ways (1, 2, 4), and scheduling policy (random, affinity), for the 16-processor configuration.

To highlight the effects of process migration on miss rate we varied the scheduling policy (random and cache affinity). Figures 3 and 4 investigate the effects in the case of 4 processor and 16 processors, respectively. Cache affinity mainly causes a reduction of the "other misses". In the 4-processor case the benefit is higher than in the 16-processor case, due to a larger number of processes compared to the number of processors. Indeed, in this condition, there is a larger number of ready-to-execute processes, so that we have a high probability that a process can execute its time-slice on the last-assigned processor. Thus, cache affinity seems to be not so effective for high-end machines. Comparing Figures 3 and 4, we also notice that in the case of 16 processors the miss rate is higher, as we expected, due to the presence of more copies of the same block. This is caused by the larger number of processors (and caches).

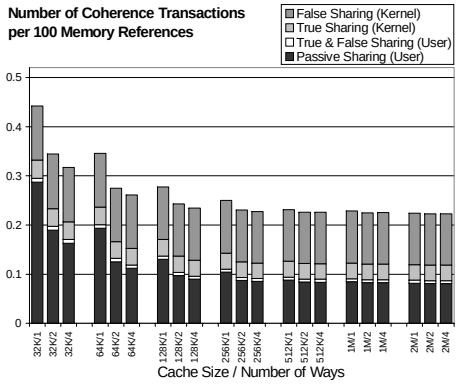

Fig. 5. Number of coherence transactions of MESI versus cache size (32K bytes, 64K bytes, 128K bytes, 256K bytes, 512K bytes, 1M bytes, 2M bytes) and number of ways (1, 2, 4), for 4-processor configuration and a random scheduling policy. The decrease of passive sharing transactions with the associativity is due to the reduction of passively shared copies produced by the write-for-invalidate transactions. That effect is more evident for small cache sizes.

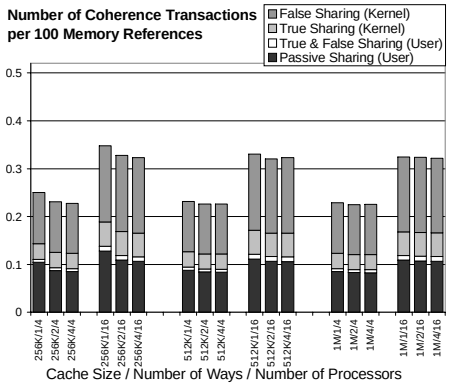

Fig. 6. Number of coherence transactions (write-for-invalidate transactions) versus cache size (256K bytes, 512K bytes, 1M bytes), number of processors (4, 16), and number of ways (1, 2, 4), for a random scheduling policy. There is an increase in each component of this overhead in the high-end (16 processor) configuration.

As observed above, cache misses produce a relevant traffic of *read-block* transactions on the bus. Thus, the graphs in Figures 1, 3, 4, can be read as the *number of read-block transactions per 100 memory references*. The most significant part of the rest of transactions is due to coherence transactions (e.g. *write-for-invalidate* transactions, in the case of MESI). Figure 5 and 6 show such coherence transactions when cache size, ways, and number of processors is varied. In order to allow the comparison between *read-block* and *write-for-invalidate* traffic, we used the same metric for the two quantities: *number of bus transactions generated by 100 references*.

As for the overhead produced by shared data accesses, the behavior seen in the previous figures is confirmed: kernel related overhead is more consistent than user-related overhead and false sharing dominates it. However, user accesses also exhibit a noticeable amount of passive sharing, produced by private data as a consequence of process migration. Passive sharing overhead decreases with larger cache sizes and with a higher number of ways. This non-intuitive result is more evident in the case of small caches, and is a consequence of two competing factors. On one hand, there is an

increase of passive shared copies due to their longer lifetime. This is produced by the increase of cache size, or set-associativity. On the other, MESI invalidates remote copies on every write-miss, thus reducing the passively shared copies.

Passive sharing increases when switching from the 4- to the 16-processor configuration (Figure 6). As it happens for false sharing, also passive sharing is a "useless" sharing, since it is consequence of migration and it is not caused by accesses to active shared data.

We considered three different techniques that could reduce passive sharing: two are based on coherence protocol (PSCR [9], and AMSD [19], [4]), and one is based on an affinity scheduling algorithm [18]. AMSD is our acronym, which stands for Adaptive Migratory Sharing Detection. 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. The protocol identifies migratory-shared data dynamically in order to reduce the cost of moving them. The implementation relies on an extension of a common MESI protocol. Coherence traffic is again due to *write-for-invalidate* transactions and *invalidation* transactions.

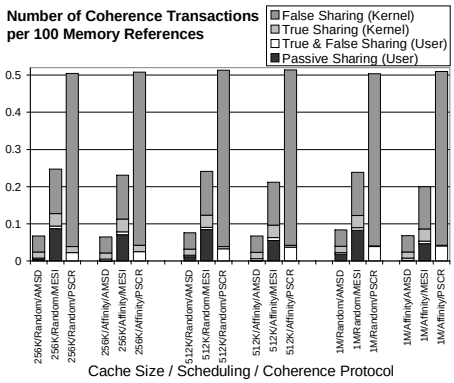

Fig. 7. Number of coherence transactions of versus cache size (256K bytes, 512K bytes, 1M bytes), scheduling algorithm (random, affinity), coherence protocol (MESI, PSCR, AMSD). Data assume 4 processors and a two-way set associative cache. Both AMSD and PSCR are effective in reducing or eliminating passive sharing overhead.

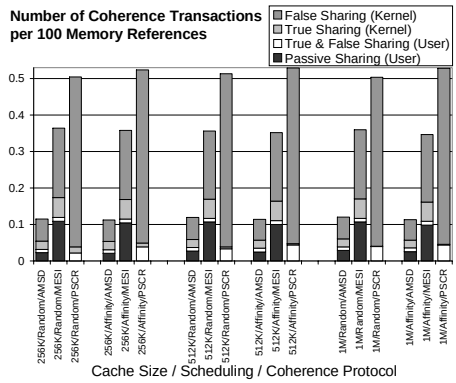

Fig. 8. Number of coherence transactions versus cache size (256K bytes, 512K bytes, 1M bytes), scheduling algorithm (random, affinity), and coherence protocol (AMSD, PSCR, MESI). Data assume 16 processors and a two-way set associative cache. The affinity scheduling produces only a slightly reduction of passive sharing.

PSCR (Passive Shared Copy Removal) adopts a selective invalidation for the private data, and uses the *write-update* scheme for the actively shared data. A cached copy belonging to a process private area is invalidated locally as soon as another processor fetches the same block.

In the case of four processors (Figure 7), the adoption of AMSD causes a reduction of passive, true, and false sharing transactions compared to the MESI case. PSCR eliminates the transactions due to passive sharing, but causes a higher number of them in the other kinds of sharing, due to the broadcast nature of the protocol. Cache affinity technique also reduces passive sharing transactions, so it does not improve PSCR performance. In the case of 16 processors (Figure 8), we have the same effects for the number of transactions, except in the case of affinity. Indeed, since the number of processors is getting closer to the number of processes, the affinity scheduling is not always applicable. The scheduler is forced to allocate a ready process on a different available processor.

Figure 9 also considers the cost of transactions combining the effects on performance in a single figure, the Global System Power (i.e. the sum of processor utilization [9]). Again, we varied the cache size, the protocol, the scheduling policy, and the number of processors. In the case of four processors, all the three techniques do not produce a sensible effect on performance. The situation is very different in the case of sixteen processors. In that case PSCR and AMSD allow much better performance than MESI. As shown in Figure 10 (and 8) this is due to a reduction of the miss rate for PSCR and to a reduction of coherence transactions for AMSD.

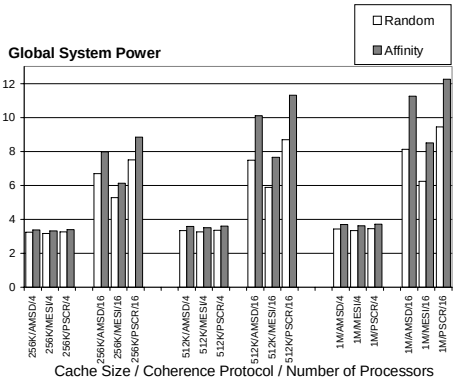

Fig. 9. Global System Power versus cache size (256K bytes, 512K bytes, 1M bytes), coherence protocol (AMSD, PSCR, MESI), and number of processors (4, 16) for two scheduling policy (random, affinity). Data assume a two way set associative cache. Global System Power is the sum of processor utilizations. In all cases PSCR and AMSD show better processor utilization than MESI.

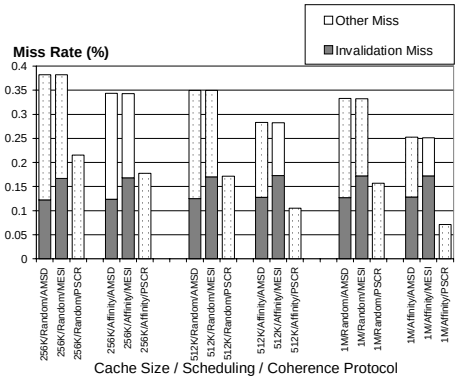

Fig. 10. Miss Rate versus cache size (256K bytes, 512K bytes, 1M bytes), scheduling policy (random, affinity), and coherence protocol (AMSD, PSCR, MESI). Data assume a 4-processor configuration, and a two way set associative cache. The miss rate is almost the same for MESI and AMSD, even if their miss component actually varies. PSCR allows a much lower miss rate compared to the other protocols' due to the absence of the invalidation miss.

5 Conclusions

In this paper we characterized the number and type of transactions induced on the shared-bus of a shared-memory multiprocessors used as a Web-Server machine. Our workload has been set up by tracing a combined workload made of an HTTP server (Apache), PostgreSQL DB-server resolving TPC-D benchmark queries, and typical UNIX shell commands. The analysis has been carried out with trace-driven simulation, and by considering not only user references, but also the most influencing kernel activities. The low usage of shared memory in the user space causes most of the data sharing to happen in the kernel space. The larger part of this sharing has resulted to be due to false sharing, and secondly to passive sharing. To eliminate false sharing, we suggest the redesign of kernel structures. We put into evidence how the MESI protocol is not capable of treating passive sharing, that is the sharing generated on private data as a consequence of process migration. These facts indicate that margins to improve MESI protocol still exist. The use of affinity scheduling algorithms determines only a partial reduction of passive sharing and this technique does not adapt to all load conditions. Ad hoc protocols, like PSCR and AMSD, better reduce the main causes of overhead and allow better performance than MESI.

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 1998.
2. M. Baentsch, L. Baum, and G. Molter, “Enhancing the Web’s Infrastructure: From Caching to Replication”. *Internet Computing*, vol. 1, no. 2, pp. 18-27, Mar.-Apr. 1997.
3. L. A. Barroso, K. Gharachorloo, and E. Bugnion, “Memory System Characterization of Commercial Workloads”. *Proc. of the 25th Intl. Symp. on Computer Architecture*, pp. 3-14, June 1998.
4. A. L. Cox and R. J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proc. 20th Intl. Symp. on Computer Architecture*, San Diego, California, pp. 98-108, May 1993.
5. J. Edwards, “The changing Face of Freeware”. IEEE *Computer*, Vol. 31, N. 10, pp 11-13, Oct. 1998.
6. K. Gharachorloo, A. Gupta, and J. Hennessy, “Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors”, Proc. of the *4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 245-357, Apr. 1991.
7. R. Giorgi, C. Prete, G. Prina, L. Ricciardi, “A Hybrid Approach to Trace Generation for Performance Evaluation of Shared-Bus Multiprocessors”. *Proc. 22nd EuroMicro Intl. Conf.*, Prague, pp. 207-241, September 1996.
8. R. Giorgi, C. A. Prete, G. Prina, L. Ricciardi, “Trace Factory: a Workload Generation Environment for Trace-Driven Simulation of Shared-Bus Multiprocessor”. *IEEE Concurrency*, 5(4), pp. 54-68, Oct-Dec 1997.
9. R. Giorgi, C. A. Prete, “PSCR: A Coherence Protocol for Eliminating Passive Sharing in Shared-Bus Shared-Memory Multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, n. 7, pp. 742-763, July 1999.
10. J. Hennessy and D.A. Petterson, *Computer Architecture: a Quantitative Approach, 2nd edition*. Morgan Kaufmann Pablishers, San francisco, CA, 1996.
11. R. L. Hyde and B. D. Fleisch, “An Analysis of Degenerate Sharing and False Coherence”. *Journal of Parallel and Distributed Computing*, 34(2), pp. 183-195, May 1996.
12. T. E. Jeremiassen and S. J. Eggers, “Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations”, *ACM SIGPLAN Notice*, 30(8), pp.179-188, Aug. 1995.
13. 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.
14. 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.
15. C. A. Prete, G. Prina, and L. Ricciardi, “A Trace Driven Simulator for Performance Evaluation of Cache-Based Multiprocessor System”. *IEEE Transactions on Parallel and Distributed Systems*, vol. 6 (9), pp. 915-929, September 1995
16. 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.
17. D. Robinson and the Apache Group, APACHE – An HTTP Server, Reference Manual, 1995. .
18. M. S. Squillante, D. E. Lazowska, “Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling”. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4 (2), pp. 131-143, February 1993.
19. P. Stenstrom, M. Brorsson, and L. Sandberg, “An Adaptive Cache Coherence Protocol Optimezed for Migratory Sharing”. *Proc. of the 20th Intl. Symp. on Computer Architecture*. San Diego, CA, May 1993.
20. 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.
21. P. Sweazey and A.J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus". *Proc. of the 13th Intl. Symp. on Computer Architecture*, pp. 414-423, June 1986.
22. M. Tomasevic and V. Milutinovic, *The Cache Coherence Problem in Shared-Memory Multiprocessors –Hardware Solutions*. IEEE Computer Society Press, Los Alamitos, CA, April 1993.
23. M. Tomasevic and V. Milutinovic, "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors". *IEEE Micro*, vol. 14, no. 5, pp. 52-59, Oct. 1994 and vol. 14, no. 6, 61-66, Dec. 1994.
24. M. Tomasevic and V. Milutinovic, “The word-invalidate cache coherence protocol”, *Microprocessors and Microsystems*, pp. 3-16, vol. 20, Mar. 1996.
25. J. Torrellas, M. S. Lam, and J.L. Hennessy, “False Sharing and Spatial Locality in Multiprocessor Caches”. *IEEE Transactions on Computer*, vol. 43, n. 6, pp. 651-663, June 1994.
26. Transaction Processing Performance Council, TPC Benchmark D Standard Specification. Dec 1995.
27. P. Trancoso, J. L. Larriba-Pey, Z. Zhang, and J. Torrellas, “The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors”. *Proc. of the 3rd Intl. Symp. on High Performance Computer Architecture*, Feb 1997.
28. R. A. Uhlig and T. N. Mudge, "Trace-Driven Memory Simulation: a survey". *ACM Computing Surveys*, pp. 128-170, June 1997.
29. A. yu and J. Chen, The POSTGRES95 User Manual. Computer science Div., dept of EECS, University of California at Berkeley, July 1995.