PERFORMANCE ANALYSIS OF ELECTRONIC COMMERCE MULTIPROCESSOR SERVER

Pierfrancesco Foglia, Roberto Giorgi, Cosimo Antonio Prete
Dipartimento di Ingegneria dell'Informazione
Facolta' di Ingegneria, Universita' di Pisa
Via Diotisalvi, 2 – 56126 PISA (Italy)
{foglia,giorgi,prete}@iet.unipi.it

Abstract

In this paper, the performance of an Electronic Commerce server, i.e. a system running Electronic Commerce applications, is evaluated in the case of shared-bus multiprocessor architecture. In particular, we focused on the memory subsystem design. We have analyzed the common case of a system using the MESI coherence protocol, for maintaining coherency among the processor private caches. We have evaluated the miss ratio and the bus traffic of such a system by varying cache size, number of ways, scheduling policy and number of processors, highlighting the relations with different types of data sharing generated by the application or the kernel. We found that passive sharing and false sharing are the major sources of coherence overhead, in the case of relatively large caches (over 1M-byte size). False sharing is mainly due to kernel data, and can be eliminated by using appropriate data structure design techniques. A scheduling technique, like cache-affinity can reduce passive sharing, but it is not effective in every load conditions. Thus, a special coherence protocol could be a better solution to completely eliminate passive sharing overhead and boost performance.

1. Introduction

Among the recent developments of the Internet, there is the integration of traditional commerce procedures: new terms have come up, like EBI (*Electronic Business over the Internet*) and E-Commerce (*Electronic Commerce*) [35], [2], [17]. Whilst, the industry has caught up the development of related hardware and software products, the academics focused on a deeper analysis of the new scenario. For example, how to cope with the problem of an adequate design of the system in order to achieve the desired performance.

To answer this kind of questions, we need to characterize better the typical architecture of Electronic Commerce applications. The common case is for three-tiered systems [15], [4], [6]: on tier one, the user machine runs a client program, typically a web-browser and Java applets; the client sends its requests to the server and receives the results to be shown to the end-user. Tier two includes a web-server that satisfies the application specific requests and takes care of the load balancing. On tier three, the service processes furnish standard services, such as DB-Management, credit-card information, catalog information, shipping information, user information, site activity log and son on. Tier two and three elements can be merged onto a single platform, or they can be distributed on several computers. The single-computer solution has the advantage of a lower cost and a simplified management. The distributed solution has flexibility, scalability, and fault-tolerance. In both cases, the systems can be based on multiprocessor architecture [26].

In the following, we shall consider an E-Commerce server based on shared-bus shared-memory multiprocessor, and in particular, we shall focus on the core architecture related problems, rather then on software, network, and I/O related issues.

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). Unfortunately, multiple cache memories introduce the coherence problem [19], [29], [30]. The coherence protocol has a great influence on the performance. Indeed, to guarantee cache coherence, the protocol needs a certain number of bus transactions (known as *coherence overhead*) that add up to the basic bus traffic of cache-based uniprocessors. Thus, a design issue is also the minimization of the coherence overhead

In this paper, we shall analyze the memory hierarchy behavior and the bus coherence overhead of a multiprocessor used as E-Commerce server. In our evaluation, tier two is constituted by the Apache daemon [22], which handles HTTP requests, tier three is

constituted by a SQL server, namely PostgreSQL [36], which handles TPC-D [33], and by several Unix utilities which both access file system and interface the various programs running on the system. The methodology relies on trace-driven simulation, by means of the "Trace Factory" environment [9], [20].

The analysis starts from a reference case, and explores different architectural choices as for cache, number of processors, and scheduling algorithm. The scheduling algorithm plays an essential role in such systems in order to obtain the load balancing among the available 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 [21], [10].

The results we obtained show that in these systems large caches and cache affinity improve the performance, in spite of the coherence-related overhead caused by large cache. Anyway, due to both false sharing and passive sharing overhead, MESI is not optimal for E-Commerce server. An accurate design of kernel structures is suggested to reduce false sharing. Special coherence protocols [10] can eliminate passive sharing overhead.

2. Performance Considerations

We considered a shared-bus, shared-memory multiprocessor architecture. The shared-bus interconnects the processor elements and the shared-memory. The design issues are speed up and system scalability. It is well known that the shared bus is the performance bottleneck, in this kind of systems. To overcome the bus limitations, and thus 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 traffic on the processor interconnection network [12], but they cause the coherence problem [19], [29], [30]. Two or more processors may store a copy of the same memory block in their private caches. When 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 updated value. The protocol activity involves a certain number of bus transactions to keep the copies coherent, which add up 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 [28], 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. Each implementation actually differs for some details, that is several flavors of MESI do exist.

We considered the implementation of MESI in the Pentium Pro and Pentium II processors [23]. That implementation can be summarized as follows. The protocol has four states: *Modified*, when the cache block holds the only updated copy that is not identical to memory block; *Exclusive*, when the cache block holds the only valid copy that is identical to the memory block; *Shared*, when the cache block holds a valid copy that is identical to the memory block, and might be also present in other caches; *Invalid*, when the cache block holds no valid information. Four different kinds of bus transaction are used: *read-block* (to fetch a block), *read-and-invalidate-block* (to fetch a block and invalidate any copies in other caches), *read-and-invalidate-for-0-bytes* (to invalidate any copies in other caches), and *update-block* (to write back dirty copies when they need to be destroyed for replacement). The fetched block can either come from another cache, or from main memory. As for the state transitions, in case of hit upon a read operation there is no state change, whilst in case of miss the new copy is loaded in the *Shared* or in the *Exclusive* state, depending on whether or not, respectively, other copies exist in the other caches. In case of write operation on a *Shared* copy, a-*read-and-invalidate-for-0-bytes* transaction is issued on the bus, in order to invalidate all remote copies. The local copy is then turned into the *Modified* state. In case of write operations involving copies in *Exclusive* or *Modified* state, the state is changed into *Modified*. In case of miss upon a write operation, a *read-and-invalidate-block* transaction is issued to load the copy and invalidate any other copy. The local copy is turned in the *Modified* state. The remote invalidations used to obtain coherency have as a drawback the need to reload the copy, if it is again used 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 and *read-and-invalidate-for-0-bytes* transactions and *Invalidation Misses.*

We also wish to relate that overhead with the kind of data sharing, in order to detect the causes for the coherence overhead. 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* [32], which occurs when several processors reference different data items belonging to the same memory block; iii) *passive* [30], [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. Whilst active sharing is unavoidable, the other two forms of sharing are useless. The relevant overhead they

produce can be reduced [31], [25], [18], and possibly avoided [10].

3. E-Commerce workload

The typical software architecture of E-commerce applications is based on a three-tiered architecture (Figure 1), which enhances the scalability and simplifies the design. According to this model, the user (or client, tier one) sends its requests by means of a web-browser or a Java applet. A daemon (tier two) waits for a user request, and sends the request to a child process, which handles the incoming request either by sending a file, in the case of a file request, or by cooperating with some service processes (tier three). Then, the daemon continues its execution and waits for new requests, while the child completes its dialog with tier three, and eventually sends back the results to the end user. We have utilized the Apache server [22] as tier two, which is currently the most popular HTTP daemon [5]. We have configured that server, so that it spawns a minimum of 2 idle processes, a maximum of 10 idle processes. The number of requests that a child processes before dying is limited to 100. Besides file transferring activity, the server might be required to use other standard applications, typically a DBMS (Data Base Management System), for carrying out a search [3], or accessing other accounting, shipping, or logging information. Our workload thus includes DBMS activity, which in our case is managed by PostgreSQL [36]. In order to obtain a general behavior not depending on a specific implementation of an E-Commerce system, the queries to the DBMS are carried out as specified by the TPC-D [33] benchmark.

PostgreSQL consists of a front-end process that accepts SQL queries, and a backend that creates processes to satisfy the queries. TPC-D simulates an application for a wholesale supplier that manages, sells and distributes a product worldwide. TPC-D data are organized in several tables; the most important are 'line-item', 'order', 'part', 'customer', and 'supplier'. The simulated company buys part (stored in 'part' table) from suppliers (stored in 'supplier' table), and sells them to the customer (stored in 'customer' table). TPC-D includes 17 read-only queries, and 2 update queries. We have populated the database up to a size of 400M bytes, by using the specific generator program distributed with the TPC-D code (dbgen). This size corresponds to 600,000 entries for the 'line-item' table and 15,000 entries for the customer table. As for other database options, we have used standard B-Tree and sequential indices.

For completing the preparation of our E-commerce workload, we considered some glue-processes that can be generated by shell scripts, or used as service processes. In a typical situation, various requests may be running, thus requiring the support of different system commands and ordinary applications. To this end, Unix utilities (ls, awk, cp, gzip, and rm) have been added to the workload.

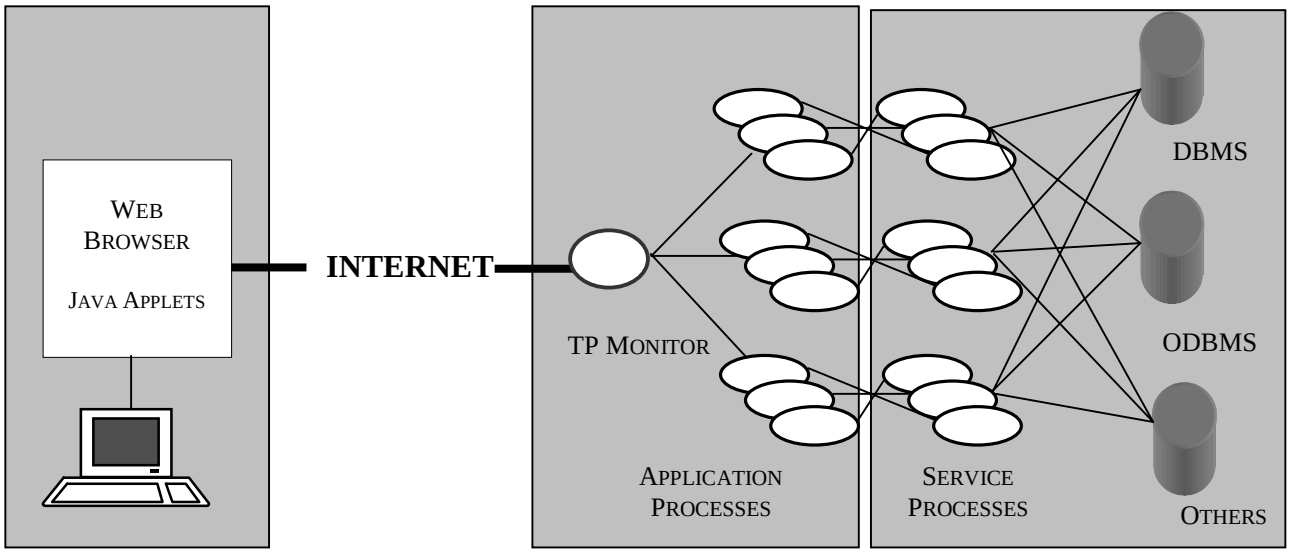

Figure 1. Typical architecture of E-Commerce application. Tier one is constituted of a client program, typically a web-browser running Java applets. Tier two is a Transaction Processing server, which includes a Transaction Processing Monitor (TP Monitor) that enables requests, spawns child application processes, and takes care of the load balance. In tier three, the service processes furnish standard services (database or other legacy applications). DBMS stands for Data Base Management System, ODBMS stands for Object-oriented Data Base Management System.

4. Performance analysis

4.1. Methodology

The methodology used in our analysis is based both on trace-driven simulation [27], [20], [34], and on the simulation of the three kernel activities that most affect performance: *system calls*, *process scheduling*, and *virtual-to-physical address translation* [8]. In the first phase, we produce a *source* trace (a sequence of user memory references, system-call positions and synchronization events in case of multiprocess programs) for each application belonging to the workload by means of a modified version of Tangolite [11]. In the second one, 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 [20], 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 [9].

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

To detect sharing patterns, and evaluate the source of overhead, we have extended an existing classification algorithm [13], by adding the evaluation of passive sharing, and extending its validity to the case of finite-size caches. Our algorithm performs on-line analysis of data access patterns, whilst the original version of this algorithm relied on off-line analysis.

Table 1. Statistics of source traces for some UNIX utilities (32-byte block size and 5,000,000 references per application)

Application	Distinct blocks	Code (%)	Data Read	Data Write
Awk (beg)	9876	76.23	14.94	8.83
awk (mid)	8129	76.13	14.88	8.99
cp	5432	77.21	13.91	8.88
gzip	7123	82.32	14.91	2.77
rm	2655	86.18	11.71	2.11
ls -aR	5860	80.23	13.98	5.79
ls -ltR (beg)	5715	78.53	14.68	6.79
ls -ltR (mid)	5091	78.22	14.18	7.60

Table 2. Statistics of multiprocess application source traces (Apache and TPC-D), in case of 32-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	188534	75.15	18.24	6.61	1105	1.52	0.49
TPC-D	5	15467	71.95	18.17	9.88	5838	2.70	0.79

The E-server workload is constituted of 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 Table 2 (for the multiprocess ones) contain some statistics of the source traces 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 3 summarizes also the statistics of the target workload used in our evaluation (E-Server). Data are related to 100 requests to the web-server (of static pages), that produce 140 millions of references.

Table 3. Statistics of workload (E-Server), in case of 32-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
E-Server	26	461810	75.49	17.12	7.39	6101	1.68	0.54

4.2. Simulation results

We considered two basic configurations: a 4-processor machine and an "high-end" 16-processor one. Each processor has a private cache whose size has been varied between 32K bytes and 2M bytes, whilst for block size we considered 32 bytes. The simulated processors are MIPS-R10000 ones; paging relays on 4-Kbyte page size; the bus supports transaction splitting, and we adopt processor-consistency memory model [7]. As base case study, a machine with 128-bit shared bus is considered. For the scheduling policy two solutions have been analyzed: random and cache-affinity; scheduler time-slice is equivalent to 200,000 references. The bus timings of base case study are summarized in Table 4.

Table 4. Numerical values of timing parameters for the multiprocessor simulator (times are in clock cycles) in the case of 32-byte block size.

Class	Parameter	Timing
CPU	Read/Write cycle	2
Bus	Read–and-Invalidate –of-0 bytes transaction	5
	Memory-to-cache read-block transaction	68
	Memory-to-cache read-and-invalidate-block transaction	68
	Cache-to-cache read-block transaction	12
	Cache-to-cache read-and-invalidate block transaction	12
	Update-block transaction	6

There are two issues in the design of a multiprocessor system: the minimization of execution time and the minimization of the bus traffic to achieve a better system scalability. To achieve these goals, the designers can optimize the memory subsystem. The miss ratio influences the waiting time of the processor, and therefore, the execution time. The bus traffic affects both the time required to serve the miss, and thus again the waiting time of the processor, and the multiprocessor

system scalability. Thus, we wish to analyze the miss ratio and bus traffic, and quantify the sources of coherence overhead.

As discussed above, in the case of MESI protocol, bus traffic has the following components: *read-block* (we assumed that *read-and-invalidate* block transactions have the same cost of *read-block* transactions*)*, *read–and-invalidate-of-0-bytes*, and *update* transactions. Update transactions are only a negligible part of the bus-traffic (very low for large cache size and, however, lower than 10% of *read-block* transactions for small cache sizes), and then they do not influence the performance greatly. Therefore, the main part of traffic is due to classical misses (sum of cold, conflict and capacity misses [12]) and coherence traffic, constituted of misses due to the invalidation of actual shared copies and *read-and-invalidate-of-0-bytes* transactions

4.2.1 Miss rate analysis. Figure 2 shows the miss rate when cache size and number of ways are varied. As expected, the number of invalidation misses (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 with more ways. For cache size above 512K bytes the difference of miss ratio between 2 and 4 ways becomes negligible. Invalidation misses are basically due to the kernel (because of the low usage of user shared data), and false sharing is the main source of this overhead (Figure 3).

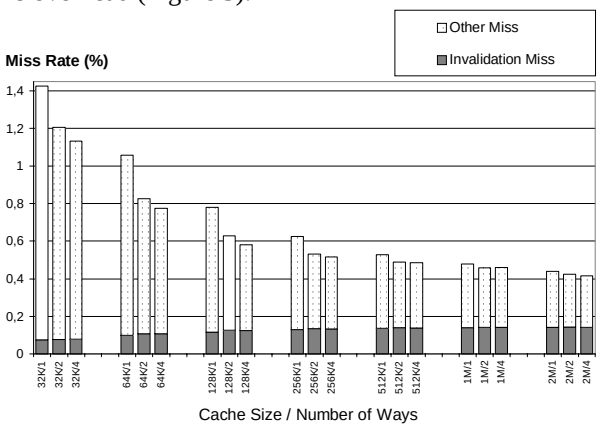

Figure 2. Breakdown of miss rate versus cache size (32K bytes, 64K bytes, 128K bytes, 256K bytes, 512K bytes, 1M bytes, 2M bytes) and number of ways (1, 2, 4), for a 4-processor system, and a random scheduling policy. "Other Miss" includes cold miss, capacity miss, and replacement miss. Miss rate decreases, whilst invalidation miss portion (i.e. the sum of false sharing misses and true sharing misses) increases with large cache size and higher associativity.

False sharing appears when data used in exclusive way by different processes become physically shared. This happens with improper data alignment, when different data objects are placed into the same block. False sharing can be eliminated either by using special coherence protocols [31], or by properly allocating the involved shared data structures [14]. 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.

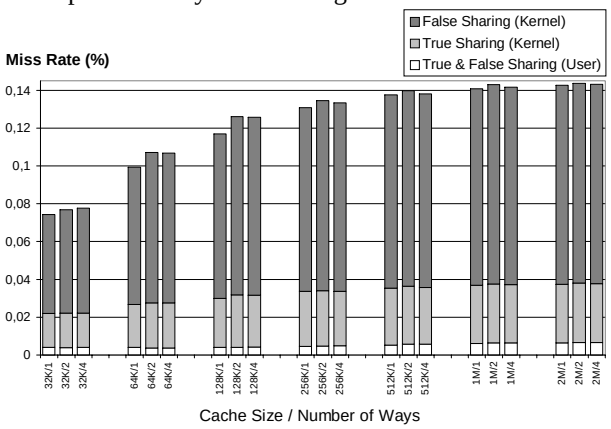

Figure 3. Breakdown of invalidation miss rate versus cache size (32K bytes, 64K bytes, 128K bytes, 256K bytes, 512K bytes, 1M bytes, 2M bytes) and number of ways (1, 2, 4), for a 4-processor system, and random scheduling policy. Invalidation misses are basically due to the kernel, and false sharing is the main source of those misses.

Invalidation miss rate is significant for large cache sizes, while is negligible for small cache sizes. Therefore, in systems with large caches (512K bytes or larger), the performance depends on the coherence overhead. For cache sizes above 256K bytes, our results indicate that the kernel of an E-commerce server should be designed as specified above in order to avoid false sharing effects. For cache sizes below 128K bytes would be more important to use design techniques which could enhance the locality of the running applications in order to minimize traditional misses [16].

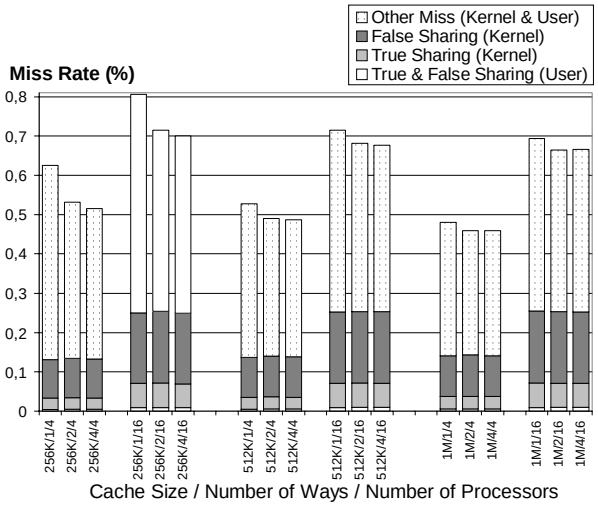

Figure 4. Breakdown of miss rate versus cache sizes (64K bytes, 256K bytes, 512K bytes, 1M bytes), number of ways (1, 2, 4) and number of processors (4, 16), in the case of random scheduling policy. The number of Invalidation misses, and particularly, false sharing misses increases on the "high end" (16-processor) configuration.

As an issue in multiprocessor design is scalability, we considered also an "high end", 16-processor configuration. The analysis of the miss rate for the 16-processor machine (Figure 4) confirms the previous results. We observe an increased contribution of invalidation misses compared with the 4-processor case, caused by the presence of more copies of the same block. This result, in turn, is a consequence of the higher number of processors (and caches).

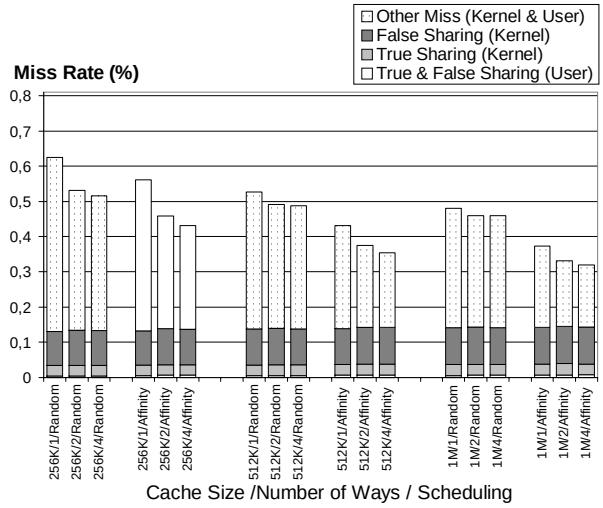

Figure 5. Breakdown of miss rate versus cache sizes (256K bytes, 512K bytes, 1M bytes), number of ways (1, 2, 4), and scheduling policy (random, affinity), for a 4-processor configuration. In case of affinity scheduler, miss rate decrease is due to the reduction of "Other Miss".

Figures 5 and 6 investigate the effects of scheduling policy. Cache affinity mainly causes a reduction of "other misses".

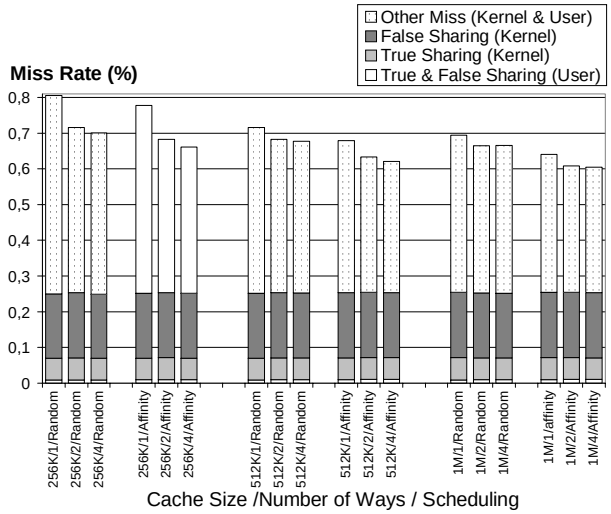

Figure 6. Breakdown of miss rate versus cache sizes (256K bytes, 512K bytes, 1M bytes), number of ways (1, 2, 4) and scheduling policy (random, affinity) for the 16-processor configuration.

In the 4-processor case (Figure 5) the benefit is higher, due to a larger number of processes compared to the number of processors. Indeed, in this condition, there is a higher number of ready-to-execute processes, so that we have a higher probability that a process can execute its time-slice on the last-used processor, and thus reuse a part of its working set.

4.2.2 Bus traffic. Misses produce a relevant traffic of *read-block* transactions on the bus. Note that every miss generates a read-block transaction on the bus; thus, the miss rate in Figures 2, 4, 5, and 6 can be considered as the number of *read-block* transactions per 100 memory references. The most significant part of the rest of bus traffic is due to *read-and–invalidate-of-0-bytes* transactions. Figure 7 and 8 show the breakdown of coherence transactions when cache size, ways, and number of processors is varied. In order to allow the comparison between *read-block* and *read-and-invalidate of-0-bytes* bus-traffic, we used the same metric for the two quantities: *number of bus transactions generated by 100 references*.

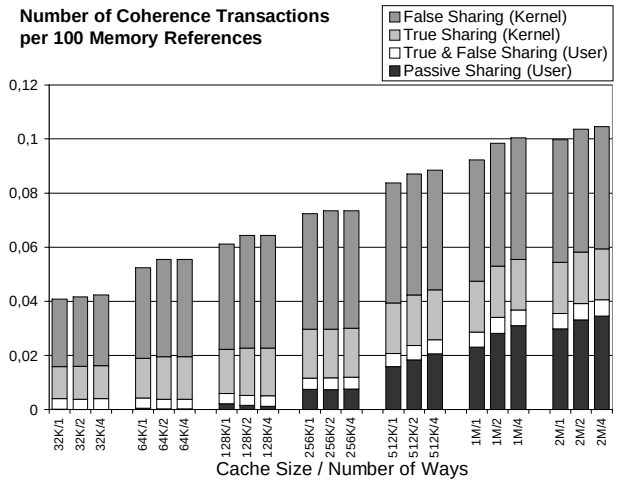

Figure 7. Number of coherence transactions *(read-and-invalidate-of-0-bytes transactions*) versus cache size (32K bytes, 64K bytes, 128K bytes, 256K bytes, 512K bytes, 1M bytes, 2M bytes) and number of ways (1, 2, 4), for a 4-processor system and a random scheduling policy. Passive sharing overhead increases with large cache and higher associativity, becoming significant in case of cache sizes larger than 256K bytes

As for the overhead produced by accesses to shared data, the behavior seen in the previous figures is again confirmed: kernel related overhead is more consistent than user-related overhead, and false sharing dominates it. However, the user accesses also exhibit a noticeable amount of passive sharing, produced by private data as a consequence of process migration. That overhead is visible for cache sizes larger than 128K bytes.
For small cache sizes that overhead is negligible due to the high replacement activity, which destroys most of the passive shared copies.

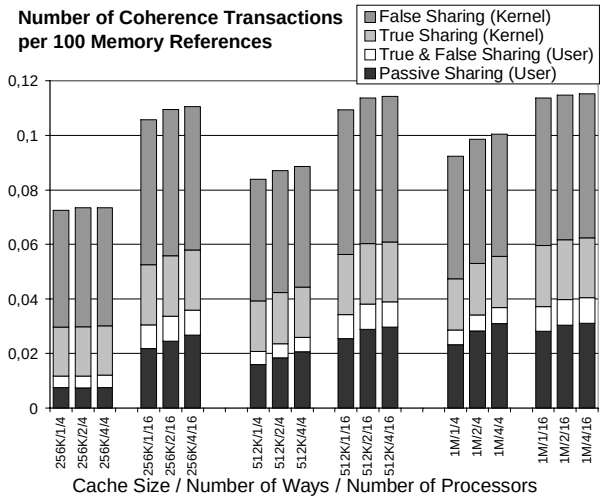

Figure 8. Number of coherence *transactions (read-and-invalidate-of-0-bytes transactions*) versus cache size (256K bytes, 512K bytes, 1M bytes), number of processors (4, 16), and number of ways (1, 2, 4), in case of a random scheduling policy. There is an increase in each component of this overhead in the "high end" (16-processor) configuration.

Passive sharing increases when switching from the 4- to the 16-processor configuration (Figure 8). As it happens for false sharing, also passive sharing is a "useless" sharing. Passive sharing is consequence of process migration, and it is not caused by accesses to shared data. Its elimination or reduction, along with the consequent bus traffic reduction, ends up in beneficial effects on the system performance. The techniques currently known that affect passive sharing are based on coherence protocols, like PSCR [10], and AMSD [25], or on affinity scheduling algorithms [24].

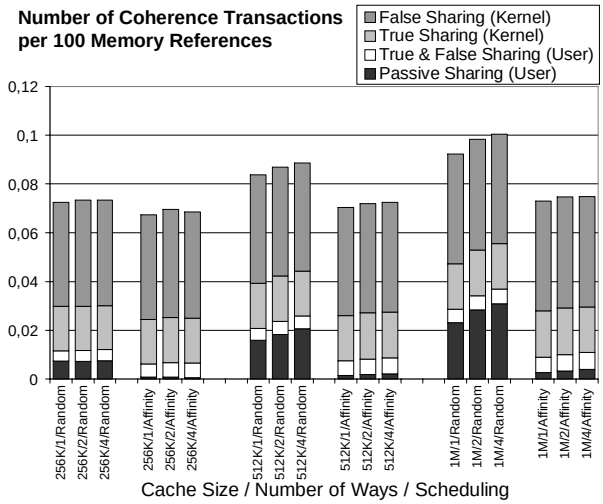

Figure 9. Number of coherence *transactions (read-and-invalidate-of-0-bytes transactions*) versus cache size (256K bytes, 512K bytes, 1M bytes), number of ways (1, 2, 4) and scheduling algorithm (random, affinity). Data assume 4 processors. Cache affinity reduces mainly passive sharing transactions; this reduction is more effective for larger cache sizes.

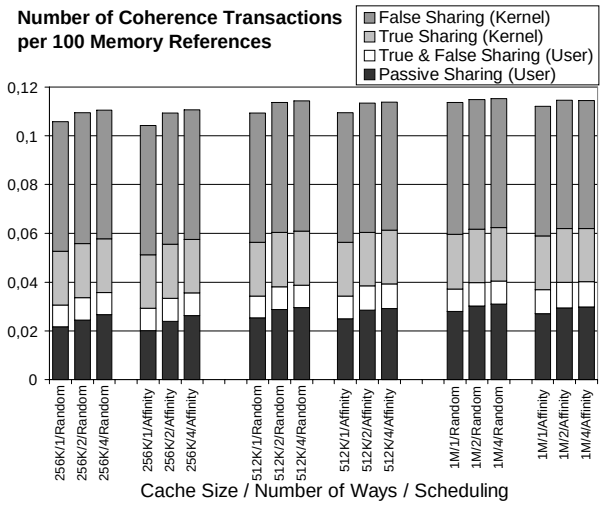

Figure 10. Number of coherence transactions (*read-and-invalidate-of-0-bytes transactions*) versus cache size (256K bytes, 512K bytes, 1M bytes), number of ways (1, 2, 4) and scheduling algorithm (random, affinity). Data assume 16 processors. In this case, affinity scheduling produces only a slightly reduction of passive sharing.

A reduction of passive sharing overhead by using cache affinity can be observed in the case of 4 processors (Figure 9). But, in the case of 16 processors (Figure 10), that reduction is negligible. 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 the ready processes on an available processor, which is different by the last-used one, thus generating a larger number of passive-shared copies. Anyway, passive sharing is the main part of the overhead induced by private data, and a consistent part of the total overhead. Both an affinity scheduling algorithm and a special coherency protocol may be used together, by obtaining the better performance in all the load conditions.

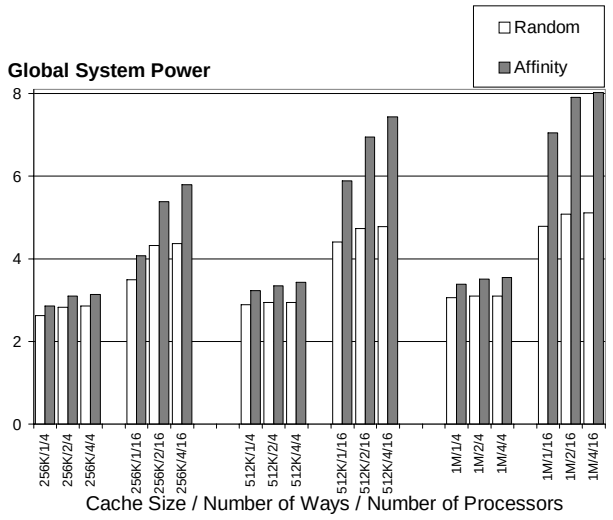

Figure 11. Global System Power versus cache size (64K bytes, 256K bytes, 1M bytes), number of ways (1, 2, 4), number of processors (4, 16), and scheduling policy (random, affinity). Global System Power is the sum of processor utilizations. In all cases, there is an increment in the utilization, when cache affinity scheduling is used. This increment is significant for cache size larger than 256K bytes and in case of 16-processor configuration.

4.2.3 Global System Power. Figure 11 considers the cost of transactions, by combining the effects on performance in a single figure, the Global System Power (i.e. the sum of processor utilization [10]). In this Figure we also compare the cases of different scheduling policies, and the 4- and 16-processor cases. In the 4-processor case, the utilization is almost near to the theoretic limit of processor utilization. The situation is different in the 16-processor case. In that case, the processor utilization is low without cache affinity. The situation is better when affinity scheduling is introduced, even if cache-affinity is not effective in reducing both the miss-rate and the coherence overhead (as shown in Figures 6 and 10). This is a consequence of the higher bus-utilization in the case of 16 processors compared to the case of 4 processors. Indeed, when the bus is more loaded, even a small reduction of its load produces a consistent reduction of the miss-cost and thus higher processor utilization. We finally point out that even if cache-affinity improves the performance, there is still much space for further performance improvement. This is due to cache affinity limitation of affecting coherence overhead in all load conditions.

The results shown in Figure 11, along with those obtained on coherence transactions (Figure 7 and 8) and on miss rate (Figures 2 to 6), indicate that we can positively use caches of sizes up to 2M bytes and cache affinity scheduling techniques, as we expected. Anyway, there is still space to improve the performance especially in the case of the high-end configuration.

5. Conclusions

In this paper, we characterized the cache misses and the bus traffic induced on the shared-bus of a shared-memory multiprocessor used as an Electronic Commerce server machine.

Our workload has been set up by considering an HTTP server (Apache), TPC-D benchmark queries, PostgreSQL DB-server, and typical UNIX shell commands. The analysis has been carried out through trace-driven simulation and by considering not only user references but also the most influencing kernel activities.

Results show that, in these systems, large caches and cache affinity can improve the performance in spite of the large coherence related overhead. In these type of applications, passive sharing and false sharing greatly affects performance. False sharing is mainly due to kernel data structures. Passive sharing is due to private data of migrating processes.

To eliminate false sharing we suggest to design the data structures in an appropriate way, as observed by others. In this case, we found that this technique can be applied more successfully for the kernel data. MESI reduces passive sharing, although it does not avoid the passive sharing overhead. The use of affinity scheduling algorithms does not allow avoiding this overhead in all load conditions. The load can vary a lot in the case of E-commerce server, so that ad-hoc protocols like PSCR and AMSD could improve the performance. Both an affinity scheduling algorithm and a special coherency protocol may be used together, by obtaining the better performance in all the load conditions.

6. Acknowledgements

The work described in this paper has been partly carried out under the financial support of the Italian “Ministero dell'Università e della Ricerca Scientifica e Tecnologica” (MURST), in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project. We thank the precious suggestions of the anonymous referees, which helped improve the quality of this paper.

7. 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] J.M. Andreoli, Francois Pacull, and R. Pareschi, “XPECT: A Framework for Electronic Commerce”, *IEEE Internet Computing*, vol. 1, no. 4, pp. 40-48, July–August 1997.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads". Proc. of the *25th International Symposium on Computer Architecture*, pp. 3-14, June 1998.

[4] R. Brandau, T. Confrey, A. D'Silva, C.J. Matheus, R. Weihmayer, "Reinventing GTE with Information Technology". *IEEE Computer*, vol. 32, no. 3, pp. 50-58, March 1999.

[5] J. Edwards, "The changing Face of Freeware". *IEEE Computer*, vol. 31, no. 10, pp. 11-13, October 1998.

[6] J. Edwards, *3-Tier Client/Server At Work*. Wiley Computer Publishing, New York, N.Y., 1999.

[7] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors". Proc. of the *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 245-357, Apr. 1991.

[8] R. Giorgi, C. Prete, G. Prina, L. Ricciardi, "A Hybrid Approach to Trace Generation for Performance Evaluation of Shared-Bus Multiprocessors". Proc. of the *22nd EuroMicro International Conference*, Prague, pp. 207-214, Sept. 1996.

[9] R. Giorgi, C. Prete, G. Prina and L. Ricciardi, "Trace Factory: a Workload Generation Environment for Trace-Driven Simulation of Shared-Bus Multiprocessor". *IEEE Concurrency*, vol. 5, no. 4, pp. 54-68, Oct-Dec 1997.

[10] 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, pp. 742-763, vol. 10, no. 7, July 1999.*

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

[12] J. Hennessy and D. A. Petterson, *Computer Architecture: a Quantitative Approach, 2nd edition*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[13] R. L. Hyde and B. D. Fleisch, "An Analysis of Degenerate Sharing and False Coherence". *Journal of Parallel and Distributed Computing*, vol. 34, no. 2, pp. 183-195, May 1996.

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

[15] T. Lewis, "The Legacy Maturity Model". *IEEE Computer*, vol. 31, no. 11, pp. 125-128, November 1998.

[16] S. Lorenzini, G. Luculli, C. A. Prete, "A Fast Procedure Placement Algorithm for Optimal Cache Use", Proc. of the *MELECON'98*, Tel Aviv, Israel, pp 1279-1284, May 1998.

[17] V. Milutinovic, *System Support for Electronic Business on Internet*. http://galeb.etf.bg.ac.yu/ ~vm/books/2001/ebi.html

[18] 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.

[19] 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.

[20] 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, no. 9, pp. 915-929, September 1995

[21] 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.

[22] D. Robinson and the Apache Group, APACHE – An HTTP Server, Reference Manual, 1995. http://www.apache.org.

[23] T. Shanley and Mindshare, Inc., *Pentium Pro and Pentium II System Architecture*, 2nd edition, Addison Wesley, Reading, MA, 1999.

[24] M. S. Squillante and D. E. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling". *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 131-143, Feb. 1993.

[25] P. Stenstrom, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimezed for Migratory Sharing". Proc. of the *20th Annual International Symposium on Computer Architecture*. San Diego, CA, May 1993.

[26] P. Stenstrom, E. Hagersten, D. J. Li Margaret Martonosi and M. Venugopal, *"Trends in Shared Memory Multiprocessing "*, *IEEE Computer*, vol. 30, no. 12 pp. 44-50, Dec. 1997.

[27] 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.

[28] P. Sweazey and A. J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus". Proc. of the *13th International Symposium on Computer Architecture*, pp. 414-423, June 1986.

[29] M. Tomasevic and V. Milutinovic, *"The Cache Coherence Problem in Shared-Memory Multiprocessors – Hardware Solutions*". IEEE Computer Society Press, Los Alamitos, CA, April 1993.

[30] 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.

[31] M. Tomasevic and V. Milutinovic, "The Word-Invalidate Cache Coherence Protocol", *Microprocessors and Microsystems*, pp. 3-16, vol. 20, Mar. 1996.

[32] J. Torrellas, M. S. Lam, and J.L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches". *IEEE Transactions on Computer*, vol. 43, no. 6, pp. 651-663, June 1994.

[33] TPC Benchmark D (Decision Support) Standard Specification. Transaction Processing Performance Council, Dec 1995.

[34] R. A. Uhlig and T. N. Mudge, "Trace-Driven Memory Simulation: a survey". *ACM Computing Surveys*, pp. 128-170, June 1997.

[35] D. W. Walker, "Free-Market Computing and the Global Economic Infrastructure", *IEEE Parallel and Distributed Technology*, Vol. 4, no. 3, pp. 60-62, FALL 1996.

[36] A. Yu and J. Chen, The POSTGRES95 User Manual. Computer Science Div., Dept of EECS, University of California at Berkeley, July 1995.