Boosting the Performance of Three-Tier Web Servers Deploying SMP Architecture

Pierfrancesco Foglia*, Roberto Giorgi+, Cosimo Antonio Prete*

* Dipartimento di Ingegneria dell'Informazione, Universita' di Pisa , Via Diotisalvi 2, 56126 Pisa, Italy
{foglia,prete}@iet.unipi.it
+ Dipartimento di Ingegneria dell'Informazione, Universita' di Siena, Via Roma 56, 53100 Siena, Italy
giorgi@unisi.it

Abstract. The focus of this paper is on analyzing the effectiveness of SMP (Symmetric Multi-Processor) architecture for implementing Three-Tier Web-Servers. In particular, we considered a workload based on the TPC-W benchmark to evaluate the system.
As the major bottleneck of this system is accessing memory through the shared bus, we analyzed what are the benefits of adopting several solutions aimed at boosting the global performance of the Web Server. Our aim is also to quantify the scalability of such a system and suggest solutions to achieve the desired processing power. The analysis starts from a reference case, and explores different architectural choices as for cache, scheduling algorithm, and coherence protocol in order to increase the number of processors possibly connected through the shared bus.
Our results show that such an SMP based server could be scaled (up to 20 processor) quite above the limits expected for this kind of architecture, if particular attention is used in solving problems related to process migration and coherence overhead.

Keywords: Multiprocessor, Shared-Memory, Coherence Protocol, Performance Evaluation, Process Migration.

1 Introduction

Web-Servers are often used as three-tier systems for E-Commerce applications [10], [21]. On tier one, the user machine runs a client program, typically a web-browser and/or Java applets; the client sends its requests to the server and receives the results to be shown to the user. Tier two includes a web-server that satisfies application specific requests, takes care of the task management and delivers standard services such as transaction management and activity log. Tier three contains data and their managers, typically DBMS systems, to furnish credit-card information, catalog information, shipping information, and user information. Tier two and three elements can be merged onto a single platform, or they can be distributed on several computers (clustered solution [26]). 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 [29].

We considered servers based on shared-bus shared-memory multiprocessor systems. In this case, design issues are scalability and speedup, which may be limited by memory latency and bus traffic. Using cache memories can reduce both. Unfortunately, multiple cache memories introduce the coherence problem [22], [32]. The coherence protocol may have 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 uni-processors. Thus, a design issue is also the minimization of the coherence overhead. A commonly adopted solution to coherence problem is the use of MESI protocol [31]. This protocol might not be performance effective for shared-bus architecture, and in particular when process migration is allowed to maintain the load balancing among processors. The scheduling algorithm plays an essential role in such systems in order to obtain load balancing. The consequent process migration generates *passive sharing*: private data blocks of a process can become resident in multiple caches. Coherence has to be enforced even on those data, but generates useless coherence overhead, which in turn may limit system performance [13], [23].

In our evaluation, the workloads have been setup as specified in the TPC-W benchmark [35]. TPC-W simulates the activities of a business-oriented transactional web server. Our aim is to quantify the scalability of such a system and suggest solutions to achieve the desired processing power. We considered the major bottlenecks of the memory system of this architecture. The results we obtained show that, by reducing the coherence overhead and the effects of process migration on the memory sub-system –acting on the affinity scheduler and the coherence protocol, - we could scale this kind of architecture up to 20 processors.

2 Related Work

Several important categories of general purpose and commercial applications, like web-server and database applications, motivated a realistic evaluation framework for shared memory multiprocessing research [29]. Several studies started to consider benchmarks like TPC-series (including DSS, OLTP, WEB-server benchmarks) representative of commercial workloads to evaluate the performance of multiprocessor servers [4], [5], [6], [19], [36].

Cain et al. [5] implemented TPC-W as a collection of Java servlets and present an architectural study detailing the memory system and branch predictor behavior of this workload. They used a 6-processor IBM RS/6000 S80 SMP machine, running AIX 4.33 operating system. They also evaluated the effectiveness of a coarse-grained multithreaded processor, simulated using SimOS, at increasing system throughput. However, their evaluation uses only no more than 6 processors. They found that the false sharing is almost absent in the user part as we also verified.

Other evaluations considered TPC-based benchmarks [4], [6], [19], [36] for database workloads. Most of the conclusions found in these evaluations present analogies with our evaluation. In particular, large caches, more associativity, and larger blocks help in the case of large working set. The major drawback of large caches is the increase coherence overhead. In our case, instead of considering single

query execution, we run multiple concurrent query streams. We consider also configurations with more processors and with different solutions for the cache parameters, coherence protocol, and scheduling policies (in particular cache affinity).

3 Web-Server Server Setup and Workload Definition

The typical software architecture of Web-Servers for e-commerce applications is based on a three-tiered model: tier one is constituted by the e-commerce clients (typically a Web Browser), which access the server by the Internet; tier two is constituted by the Web Server, a transaction management process and the application processes (which also provide accounting and auditing); tier three is constituted by data and their managers [10].

The activity of e-commerce systems typically involves data scan (to access product list, product features, credit card information, shipping information), update (to update customer status and activity status) and transactions (for instance to buy products, make payments). These activities involve the interaction between tiers according to the following model: the user (i.e. the client, tier one) sends its requests by means of a Web-Browser. A process (a daemon, which constitutes part of the tier two) waits for the user request, and sends the request to a child application process. Then, the daemon waits for new requests, while the child process handles the incoming request. This activity may require accessing html pages and/or invocating service processes at tier three.

As for workloads, we implemented a software architecture based on the following freeware components. The system front-end (part of the tier two) includes an Apache Server [24] (which is currently the most popular HTTP server [9]) Client requests that involve database activities are forwarded, via CGI interface, to the Data-Base Management System (DBMS) PostgreSQL [38]. PostgreSQL is constituted by a front-end (also part of the tier two), which intercepts requests and by a backend (part of the tier three), which executes the queries.

We configured the Apache server, so that it spawns a minimum of 8 idle processes, a maximum of 40 idle processes. The number of requests that a child can process before dying is limited to 100. PostgreSQL utilizes shared memory to cache frequently accessed data, indices, and locking structures [36].

We considered general cases of workloads not depending on the specific system. To this end, we setup the experiments as described in the TPC-W benchmark [35], which specifies how to simulate the activities of a business-oriented transactional web server and exercises the breadth of system components. The application portrayed by the benchmark is a retail store with customer browse-and-order scenario.

In a typical situation, application and management processes can require the support of different system commands and ordinary applications. To this end, Unix utilities (`ls`, `awk`, `cp`, `gzip`, and `rm`) have been considered in our workload setup. These utilities [14] are important because: i) they increase the effects of process migration as discussed in detail in the Section 5; ii) they may interfere with the shared data and code footprint of the other applications.

4 Methodology and Hardware System Configuration

The methodology used in our performance evaluation is based both on trace-driven simulation [30], [37] and on the simulation of the three kernel activities that most affect performance: *system calls*, *process scheduling*, and *virtual-to-physical memory address translation*. We used the Trace Factory environment [12]. The approach used in this environment is to produce a process trace (a sequence of user memory references, system-call positions and synchronization events in case of multi-process programs) for each process belonging to the workload by means of a modified version of Tangolite [15]. By using this tool, we have also traced the system calls of a Linux kernel 2.2.13 [20].

Process scheduling is modeled by dynamically assigning 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 multi-process applications. Virtual-to-physical memory address translation is modeled by mapping sequential virtual pages into non-sequential physical pages.

By using this methodology, the TPC-W benchmark specification, and the freeware components, we generated our target workload. We traced the execution of the workload programs handling of 100 web interactions in a specific time interval corresponding to 130 millions of references.

Table 1. Statistics of source traces for some UNIX utilities, in case of 32-byte block size

Application	Distinct blocks	Code (%)	Data (%)	Data Write (%)
AWK	9876	76.23	23.77	8.83
CP	5432	77.21	22.79	8.88
GZIP	7123	82.32	17.68	2.77
RM	2655	86.18	13.82	2.11
LS -AR	5860	80.23	19.77	5.79

Table 2. Statistics of multi-process application source, in case of 32-byte block size

Number of processes	Distinct Blocks	Code (%)	Data (%)		Shared blocks	Shared data (%)	
			Access	Write		Access	Write
8 (PostgreSQL)	24141	71.94	28.06	9.89	5838	2.70	0.79
13 (Apache)	34311	73.84	26.16	6.99	1105	1.84	0.60

The target workload is constituted of 13 processes spawned by the Apache daemon, 8 by PostgreSQL, and 5 Unix utilities. Table 1 (for the uniprocess applications) and Table 2 (for the multi-process ones) contain some statistics of the traces used to generate the workloads for a 32-byte block size. Table 3 summarizes the statistics of the resulting workloads.

Table 3. Statistics of target workload, in case of 32-byte block size

Number of processes	Distinct blocks	Code (%)	Data (%)		Shared blocks	Shared data (%)	
			Access	Write		Accesses	Write
26	112183	75.49	17.12	7.39	6101	1.68	0.54

The simulator of Trace Factory characterizes a shared-bus multiprocessor in terms of CPU, cache, and bus parameters. The CPU parameters are the number of clock cycles for a read/write CPU operation. The simulated processors are MIPS-R10000 ones; paging relays on 4-Kbyte page size. The cache parameters are cache size, block size, and associativity. The caches are non-blocking ones using a LRU (Least Recently Used) block replacement policy. We assumed a constant cache access time by the processor for all configurations.

Each processor uses a write buffer thus implementing a relaxed model of memory consistency, in particular the processor consistency model [1], [17]. Finally, the bus parameters are the number of CPU clock cycles for each kind of transaction: write, invalidation, update-block, memory-to-cache read-block, cache-to-cache read-block, memory-to-cache read-and-invalidate-block, and cache-to-cache read-and-invalidate-block. The bus supports transaction splitting.

We considered in our analysis three coherence protocols: MESI [31], AMSD [8], [28] and PSCR [13], which we describe here briefly. Although MESI is considered the industry standard, we added for comparison other two coherence protocols that perform better than MESI, in order to widen our view of possible solutions that could be combined to enhance the performance of a TPC-W workload.

Besides classical MESI protocol states, our implementation of MESI [25] uses the following bus transactions: *read-block* (to fetch a block), *read-and-invalidate-block* (to fetch a block and invalidate any copies in other caches), *invalidate* (to invalidate any copies in other caches), and *update-block* (to write back dirty copies when they need to be destroyed for replacement). The invalidation transaction used to obtain coherency has, as a drawback, the need to reload a certain copy, if a remote processor uses again that copy, thus generating a miss (*Invalidation Miss)*. Therefore, MESI coherence overhead (that is the transactions needed to enforce coherence) is due both to *Invalidate Transactions* and *Invalidation Misses*. SMP architectures based on MESI have been extensively analyzed in the case of scientific, engineering, DBMS and web workloads.

AMSD is designed for Migratory Sharing, which is a kind of true sharing that is characterized by the exclusive use of data by a certain processor for a long time interval. The protocol identifies migratory-shared data dynamically, in order to reduce the cost of moving them. Although designed for migratory sharing, AMSD may have some beneficial effects also on passive sharing. AMSD coherence overhead is due to invalidate transactions and invalidation misses.

PSCR (Passive Shared Copy Removal) adopts a selective invalidation scheme for the private data, and uses the *write-update* scheme for the shared data. A cached copy belonging to a process private area is invalidated locally as soon as another processor fetches the same. This technique eliminates passive sharing overhead. Invalidate transactions are eliminated and coherence overhead is due to write transactions.

The most significant metric for our evaluation of the machine is the GSP (Global System Power) [3], [12], which includes the combined effects of processor architecture and memory hierarchy. We recall the definition GSP:

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

where

$$U_{cpu} = (T_{cpu}-T_{delay})/T_{cpu}$$

T_{cpu} is the time needed to execute the workload, and T_{delay} is the total CPU delay time due to memory operation.

We used the miss rate to identify the main sources of memory overhead. The simulator classifies also the coherence overhead by analyzing the access patterns to shared data (true [33], false [33], e passive sharing [2], [23]). In particular, it classifies coherence transactions (write or invalidate) and misses due to a previous invalidate transaction (invalidation misses). The type of access pattern to the cache block determines the type of the coherence transaction or invalidation-miss. The classification [11] is based on an existing algorithm [18], extended to the case of passive sharing, finite size caches, and process migration.

Table 4. Timing parameters for the multiprocessor simulator (in clock cycles)

Class	Parameter	Timing			
		32 bytes	64 bytes	128 bytes	256 bytes
CPU	Read/Write operation	2	2	2	2
Bus	Invalidate transaction	5	5	5	5
	Write transaction	5	5	5	5
	Memory-to-cache read-block transaction	68	72	80	96
	Memory-to-cache read-and-invalidate-block transaction	68	72	80	96
	Cache-to-cache read-block transaction	12	16	24	40
	Cache-to-cache read-and-invalidate-block transaction	12	16	24	40
	Update-block transaction	8	12	20	36

5 Simulation Results

Our aim in this section is to show our quantitative data on solutions that could enhance the performance of a shared-bus multiprocessor utilized to as a three-tier web-server. For this reason, we first considered a reference case study and we varied the parameters that influence mostly the performance. Thus, we show data from our simulations showing how much system power we can get (in terms of GSP), and which are the hardware/software choices that we could adopt in order to build a more powerful machines.

Let us consider the Web-Server workload running on a single multiprocessor as reference case study. We considered a 128-bit shared bus. For the scheduling policy two solutions have been analyzed: random and cache-affinity [34]; scheduler time-slice is equivalent to about 200,000 references. The bus timings relative to these case studies are reported in Table 4.

The GSP graph (Figure 1) shows, as expected, that we can obtain a more powerful machine by increasing the cache size and associativity. The larger are the caches, the more scalable is the machine.

In the following we shall use this definition of scalability: we say that a multiprocessor system is scalable up to N processor, if N is the number of processors that causes the GSP to drop by more than 0.5 when the when switching between a N- to (N+1)-processor machine (this definition is equivalent to the definition of ‘critical

point' in [13]) By using this definition, the machine we are considering is scalable up to 4 processors in the case of 128-Kbyte direct access cache, and up to 9 processors in the case of 2-Mbyte 4-way caches. The higher scalability is essentially due to lower bus utilization when adopting larger caches (Figure 1). As the cache size and associativity increase, the reduction of bus traffic is due to the lower miss.

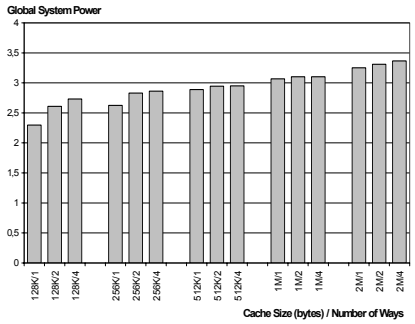

Fig. 1. Global System Power (GSP) versus cache size (128 Kbytes, 256 Kbytes, 512 Kbytes, 1 Mbytes, 2 Mbytes) and number of ways (1, 2, 4), for a 4-processor system, random scheduling policy, and 32-byte block size. The sum of processor utilizations (GSP) switches from 2.2 to 3.4 as the cache size and associativity increases

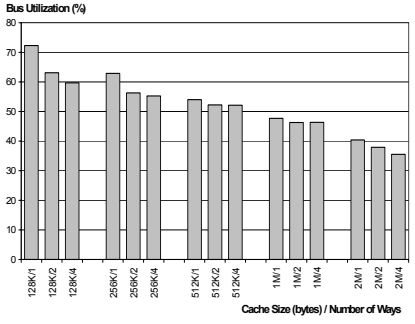

Fig. 2. Bus Utilization (in percentage) versus cache size (128 Kbytes, 256 Kbytes, 512 Kbytes, 1 Mbytes, 2 Mbytes) and number of ways (1, 2, 4), for a 4-processor system, random scheduling policy, and 32-byte block size. The less the bus utilization, the more system power can be gained by adding new processors to the system

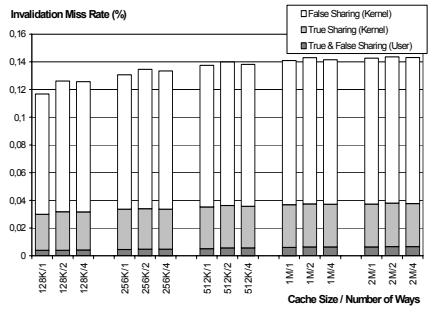

Fig. 3. Breakdown of invalidation miss rate versus cache size (128 Kbytes, 256 Kbytes, 512 Kbytes, 1 Mbytes, 2 Mbytes) and number of ways (1, 2, 4), for a 4-processor system, random scheduling policy, and 32-byte block size. Invalidation misses are basically due to the kernel, and false sharing is the main source of those misses. They slightly increase with cache size

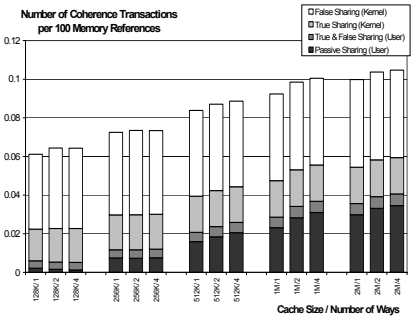

Fig. 4. Number of coherence transactions (*invalidate transactions*) versus cache size (same conditions as previous figure). Passive sharing overhead increases with cache and associativity, becoming significant in case of cache sizes larger than 256 Kbytes. Passive sharing increases due to a larger average lifetime for a cached copy when the cache size is increased

Coherence overhead increases with the cache size and associativity (Figures 3 and 4), and it weighs, in percentage, more and more on the performance. Indeed, the coherence overhead, in terms of bus utilization, changes from about 5%, in the case of 128-Kbyte direct access cache, up to 20% in the case of 2-Mbyte 4-way set associative cache. In this case, most of the coherence overhead (Figure 3) is due to false sharing generated in the kernel. True sharing is present in the kernel, whilst it is limited in the application user area. Passive sharing increases as the cache capacity is increased (Figure 4), since average lifetime of cached copies increases as well. This also shows that even a low sharing may have an important impact on the global performance: this will be more clear as the number of processors is scaled up, as we shall discus in the following paragraph. The importance of coherence traffic on performance as the cache size increases has been also highlighted in previous studies [7], [19].

As a second step, we compared the 4- and the 8-processor case. We discuss briefly the results: we found that the 8-processor configuration is near the scalability limit of the machine. In fact, the bus utilization in the 8-processor case is very high (more than 90%). Despite the fact that the GSP increases, due to higher number of processors, each processor has a lower utilization. This is essentially due to the increased bus latency.

By acting only on cache size and associativity, however, we cannot significantly increase the scalability of the machine. Thus we considered other optimizations. In particular, we can increase the performance of the 8-processor system by intervening: i) on the classical misses (sum of cold, conflict and capacity misses), ii) on the kernel false sharing, iii) by limiting the effects of process migration.

We can intervene on the point i) and ii) by modifying the block size. As for the effects on performance caused by the process migration, we can modify both the scheduling policy and the coherence protocol.

As the block size increases, we observed lower bus utilization in all the cache configurations, and a higher GSP. This is due to the decrease of miss rate, in particular due to the reduction of “classical” misses. Anyway, increasing block size become soon not so effective for block sizes above 128 bytes. In fact we should consider both higher transaction cost and coherence overhead. As we increase the block size from 32 bytes to 256 bytes, in case of 2-way caches and 2-Mbyte cache sizes, GSP increases from 5.6 to 7, and bus utilization decreases from 71% to 43%. This allows us to increase architecture scalability up to 18 processors, corresponding to a 14.5 GSP value.

This technique has the disadvantage of a higher cost to transfer the block on the bus. Another drawback is false sharing overhead, which varies with the block size. Moreover, intervening on the block size connects system performance to the program locality more tightly. Considering that program locality may vary, it is not convenient to use too much larger block sizes. Therefore, we considered in the following the results related to a system having a 128-byte block size.

As a further step, we analyzed the system when the kernel adopts a scheduling policy based on cache affinity [27]. Cache affinity produces ‘other miss’ (i.e. classical misses) rate reduction (Figure 6), and in particular the reduction of context-switch miss portion. Also, the coherence transactions are lower (Figure 7), due to the

reduction of number of passive sharing related transactions. As for invalidation misses (not reported in figure), there is no substantial difference compared to the base scheduling policy case.

In the case of 2-Mbyte, 2-way set associative cache, we observed a miss rate reduction, which causes a GSP increase from 6.9 to 7.4 (Figure 5) and a bus utilization change from 47% to 32%. This situation would allow us to extend the number of system processors up to 14, with a related increase of GSP equal to 11.3. Cache affinity reduces context switch misses, while still tolerating process migration for load balance.

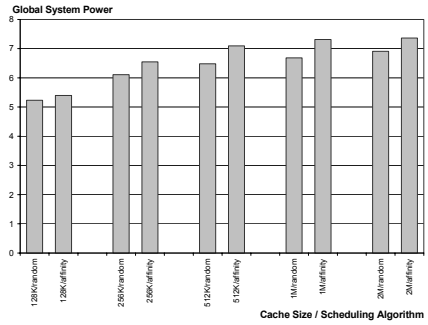

Fig. 5. Global System Power versus cache size (128 Kbytes, 256 Kbytes, 512 Kbytes, 1 Mbytes, 2 Mbytes) and scheduling policy (random, affinity), in case of 8 processors, 128-byte block size, and two-way set associative cache

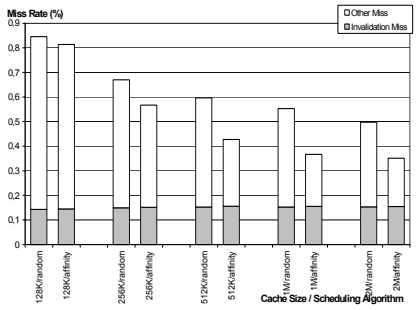

Fig. 6. Breakdown of miss rate versus cache size (128 Kbytes, 256 Kbytes, 512 Kbytes, 1 Mbytes, 2 Mbytes) and scheduling policy (random, affinity), in case of 8 processors, 128-byte block size, and two-way set associative cache. Miss reduction due to cache affinity technique is evident

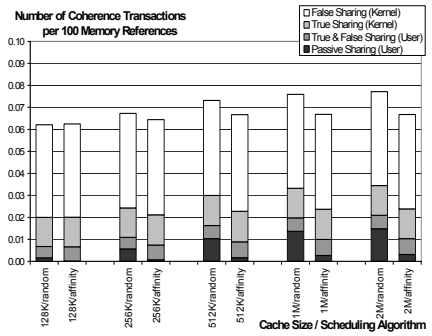

Fig. 7. Number of coherence transactions *(invalidate transactions*) versus cache size (128 Kbytes, 256 Kbytes, 512 Kbytes, 1 Mbytes, 2 Mbytes) and scheduling policy (random, affinity), in case of 8 processors, 128-byte block size, and two-way set associative cache. Cache affinity scheduling reduces only passive sharing component, while the other components remain constant

Based on our experiments, the best way to further increase the system scalability is to reduce the coherence overhead by adopting a special coherence protocol, as we show below.

We considered two additional coherence protocols that reduce or eliminate passive sharing. The first is based on Write-Update technique and the second on a Write-Invalidate technique. They are respectively, PSCR [13] and AMSD (Adaptive Migratory Sharing Detection) [8], [28]. We described briefly these protocols in Section 2.

As shown (Figure 8), as the number of processors increases, the performance difference among protocols becomes more evident. In particular, the choice of MESI protocol appears the most penalizing. This is due to the non-selective invalidation technique of MESI.

AMSD has beneficial effects on passive sharing although it does not eliminate it completely. The benefits on passive sharing are due mainly to a decrease of coherence transactions (Figure 11). The reduction of coherence transaction number is due to the behavior of AMSD on shared copies. When AMSD detects a block that has to be treated exclusively for a long time interval, it invalidates the copy locally during the handling of a remote miss, thus avoiding a necessary consequent bus transaction.

PSCR is based on the update of effectively shared copies, thus minimizing invalidation misses. By using the write-update technique, the number of coherence transactions results higher compared to other protocols (Figure 11). On the other side, the reduction of total number of misses produces a more consistent bus utilization decrease than with the other protocols. Moreover, the cost of the coherence overhead is somewhat limited by the lower cost of the coherence maintaining write transactions (Table 4). Finally, the write transaction cost is independent from the block size. More generally, in non-technical workloads has been noticed that there is a scarce reuse of data and there are large working sets [16]. This will give further advantage to such solutions that are based on write-update techniques, like PSCR.

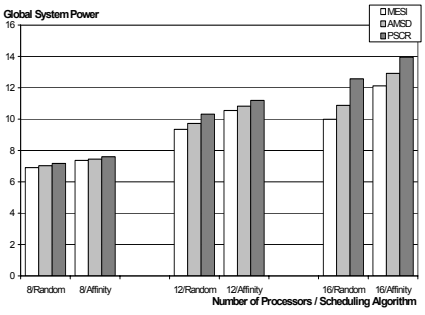

Fig. 8. Global System Power versus number of processors (8, 12, 16) and scheduling algorithm (random, affinity). Cache is a 2-way set associative with 128-byte block size, and 2-Mbyte size

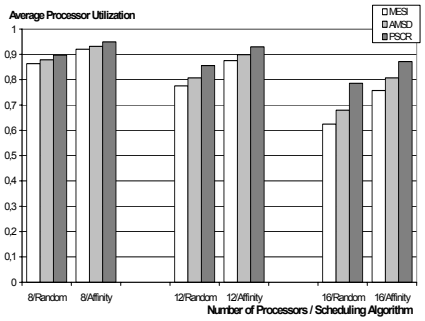

Fig. 9. Average Processor Utilization versus number of processors (8, 12, 16) and scheduling algorithm (random, affinity). Cache is a 2-way set associative with 128-byte block size, and 2-Mbyte size

Let us now analyze the scalability offered by the various protocols. As observed previously, the system is in saturation when the GSP does not increase of a minimal quantity as the number of processors is increased. In our experiments we calculated that this minimal quantity is equal to a GSP of 0.5 for each added processor. As a rule of thumb, this corresponds to a GSP increase of 2 when switching among different configurations in Figure 8. Thus, as shown in Figure 8, we can state that MESI (in case of random scheduling policy) is already near the saturation threshold for a 12-processor configuration. AMSD performs slightly better since the saturation is reached for some number of processors between 12 and 16, for both scheduling policies. In the shown configurations, PSCR is never in saturation. This justifies its adoption when higher performance (GSP) is needed.

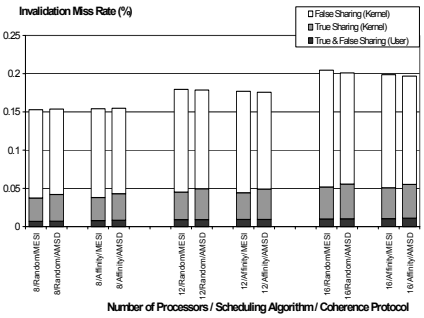

Fig. 10. Invalidation Miss Rate versus number of processor (8, 12, 16) and scheduling algorithm (random, affinity). The cache has a 128-byte block size, 2-Mbyte size, and it is 2-way set associative

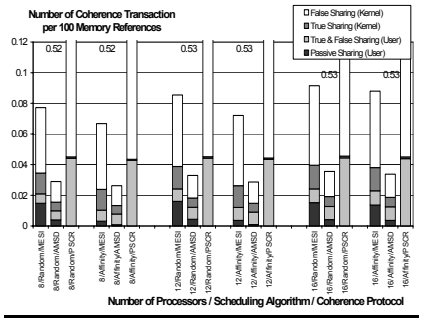

Fig. 11. Miss Rate versus number of processor (8, 12, 16) and scheduling algorithm (random, affinity). The cache has 128-byte block size, 2Mbyte size, and it is 2-way set associative

When the performance is pushed to the limits (and consequently the system works near saturation) the designer should take advantage of more optimization techniques like smart coherence protocols. The combination of all analyzed techniques (adequate block size, cache affinity, and PSCR) allows us to push system scalability up to 20 processors with a corresponding GSP of about 16.

In Table 5, we report a summary of the configuration that we tested, and how effective the solutions were, in increasing the scalability of a machine.

Table 5. Scalability that can be reached on our shared-bus multiprocessor

SYSTEM PARAMETER	NUMBER OF PROCESSORS	4	8	8	8	16
	CACHE CAPACITY (BYTES)	2M	2M	2M	2M	2M
	CACHE BLOCK SIZE (BYTES)	32	32	256	128	128
	SCHEDULING POLICY	RANDOM	RANDOM	RANDOM	AFFINITY	AFFINITY
	COHERENCE PROTOCOL	MESI	MESI	MESI	MESI	PSCR
PERFORMANCE	GSP	3.3	5.6	7	7.4	14
	BUS UTILIZATION	38%	71%	43%	32%	55%
SCALABILITY	MAX NUMBER OF PROCESSORS	9	9	18	14	20
	CORRESPONDING (ESTIMATED) GSP	~6	~6	~14	~11	~16

6 Conclusions

In this paper we analyzed a three-tier Web-Server used for e-commerce applications (TPC-W benchmark), based on a shared-bus multiprocessors SMP architecture. We analyzed what are the benefits of adopting several solutions aimed at reducing the major bottlenecks in this kind of architecture. In particular, we have analyzed in detail the memory subsystem, whose performance depends heavily on the miss rate and traffic on the shared-bus. We tried to quantify the scalability of such a system and suggested solutions to achieve the desired processing power.

As the number of processors increases, the goal of reducing coherence overhead and bus traffic becomes essential, in order to achieve good performance. When designing Web-Servers for e-commerce applications as well as other processing power demanding applications, the first goal is the reduction of classical misses. This can be achieved by using techniques that enhance the locality of the program, and other traditional solutions. Then, kernel designers should take into account false sharing. False sharing misses have to be reduced by using kernel data restructuring techniques. This could be easily achieved, since the kernel is a well-know part of the system at design time.

The use of cache affinity scheduling produces also good results for reducing classical misses and passive sharing overhead, even if its applicability is somewhat limited by the load conditions (and in particular, by the difference between number of processors and number of processes). As for architectural aspects, in the case of bus-based multiprocessors, MESI protocol is sufficient for configurations having a not so high number of processors (8 in our experiments). If a higher computing power is needed, the increase of number of processors really produces benefits, if other miss reduction techniques are considered. Coherence protocols like PSCR and AMSD, produce performance benefits. In particular PSCR eliminates coherence overhead due to passive sharing, without generating useless invalidation misses, and thus achieve better results. The adoption of PSCR allows us to extend the multiprocessor scalability at least up to 20 processors when we choose also cache affinity scheduling for the experiments that we carried out.

References

1. S.V. Adve and K. Gharachorloo: Shared Memory Consistency Models: A Tutorial. IEEE Computer, pp. 66-76, December 1996.
2. A. Agarwal and A. Gupta: Memory Reference Characteristics of Multiprocessor Applications under Mach. Proc. ACM Sigmetrics, Santa Fe, NM, pp. 215-225, May 1988.
3. J.K. Archibald and J. L. Baer: Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. ACM Transactions On Computer Systems, vol. 4, pp. 273-298, April 1986.
4. L.A. Barroso, K. Gharachorloo, and E. Bugnion: Memory System Characterization of Commercial Workloads. Proc. 25th Int. Sympo. on Computer Architecture, Barcelona, Spain, pp. 3-14, June 1998.
5. T. Cain, R. Rajwar, M. Marden, and M. Lipasti: An Architectural Characterization of Java TPC-W. 7th International Symposium of High-Performance Computer Architecture, pp. 229-240, January 2001.
6. Q. Cao, J. Torrellas, et al.: Detailed characterization of a quad Pentium Pro server running TPC-D**.** International Conference on Computer Design, pp.108-115, October 1999.
7. J. Chapin, et al.: Memory System Performance of UNIX on CC-NUMA Multiprocessors. ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems, pp. 1-13, May 1995.

8. A. L. Cox and R.J. Fowler: Adaptive Cache Coherency for Detecting Migratory Shared Data. Proc. of 20th International Symposium on Computer Architecture, San Diego, CA, pp. 98-108, May 1993.
9. J. Edwards: The changing Face of Freeware. IEEE Computer, vol. 31, no. 10, pp. 11-13, October 1998.
10. J. Edwards: 3-Tier Client/Server At Work. Wiley Computer Publishing, New York, NY, 1999.
11. P. Foglia: An Algorithm for the Classification of Coherence Related Overhead in Shared-Bus Shared-Memory Multiprocessors. IEEE TCCA Newsletter, pp. 53-58, January 2001.
12. R. Giorgi, C.A. Prete et al.: Trace Factory: a Workload Generation Environment for Trace-Driven Simulation of Shared-Bus Multiprocessor. IEEE Concurrency, vol. 5, no. 4, pp. 54-68, October 1997.
13. R. Giorgi and 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.
14. GNU Free Software Foundation. http://www.gnu.org/software/
15. S.R. Goldschmidt and J.L. Hennessy: The Accuracy of Trace-Driven Simulations of Multiprocessors. Sigmetrics Conf. on Measurement and Modeling of Computer Systems, CA, pp. 146-157, May 1993.
16. A. M. Griffazzi Maynard et al.: Contrasting characteristics and cache performance of technical and multi-user commercial workloads. Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 158-170, October 1994.
17. J. Hennessy and D.A. Patterson: Computer Architecture: a Quantitative Approach, 2nd edition. Morgan Kaufmann Publishers, San Francisco, CA, 1996.
18. 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.
19. K. Keeton, D. Patterson et al.: Performance characterization of a quad Pentium Pro SMP using OLTP workloads. Proc. of the 25th International Symposium on Computer Architecture, pp. 15-26, June 1998.
20. Linux on SGI/MIPS, http://oss.sgi.com/mips/
21. V. Milutinovic: Infrastructure for Electronic Business on the Internet. Kluwer Publishers, 2001.
22. C.A. Prete: RST Cache Memory Design for a Tightly Coupled Multiprocessor System. *IEEE Micro*, vol. 11, no. 2, pp. 16-19, 40-52, April 1991.
23. 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, March 1997.
24. D. Robinson: APACHE – An HTTP Server. Reference Manual, 1995, http://www.apache.org
25. T. Shanley and Mindshare Inc.; Pentium Pro and Pentium II System Architecture, 2nd edition. Addison Wesley, Reading, MA, 1999.
26. R. Short, R. Gamache, et al.: Windows NT Clusters for Availability and Scalability. In Proceedings of the 42nd IEEE International Computer Conference, pp. 8-13, San Jose, CA February 1997.
27. 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, February 1993.
28. P. Stenström, M. Brorsson, and L. Sandberg: An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. 20th Int. Symposium on Computer Architecture, San Diego, CA, May 1993.
29. P. Stenström, E. Hagersten, D.J. Li, M. Martonosi, and M. Venugopal. Trends in Shared Memory Multiprocessing. IEEE Computer, vol. 30, no. 12 pp. 44-50, December 1997.
30. C.B. Stunkel, B. Janssens, and W.K. Fuchs: Address Tracing for Parallel Machines. IEEE Computer, vol. 24, no. 1, pp. 31-45, January 1991.
31. P. Sweazey and A.J. Smith: A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. Proc. of the 13th Intnl. Symph, on Computer Architecture, pp. 414-423, June 1986.
32. M. Tomasevic and V. Milutinovic: The Cache Coherence Problem in Shared-Memory Multiprocessors – Hardware Solutions. IEEE Computer Society Press, Los Alamitos, CA, April 1993.
33. 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.
34. J. Torrellas et al.: Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. Journal of Parallel and Distributed Computing, vol. 24, no. 2, pp. 139-151, Feb. 1995.
35. TPC BENCHMARK W (Web Commerce) Specification, version 1.0.1. Transaction Processing Performance Council, February 2000.
36. P. Trancoso, et. al.: Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. 3rd Int. Symp. on High Perf. Computer Architecture, pp. 250-260, February 1997.
37. R.A. Uhlig and T.N. Mudge: Trace-Driven memory simulation: a survey. ACM Computing Surveys, pp. 128-170, June 1997.
38. A. Yu and J. Chen: The POSTGRES95 User Manual. Computer Science Div., Dept. of EECS, University of California at Berkeley, July 1995.