BUS UTILIZATION ANALYSIS OF MULTITHREADED SHARED-BUS MULTIPROCESSORS: INITIAL RESULTS

ROBERTO GIORGI, PIERFRANCESCO FOGLIA, COSIMO ANTONIO PRETE
Dipartimento di Ingegneria della Informazione
Università di Pisa, PISA, Italy
{giorgi,foglia,prete}@iet.unipi.it

Abstract – *A shared-bus shared-memory multiprocessor based on multithreaded CPUs is evaluated against different solutions for cache and coherence protocols. Multithreaded architectures have been intensively studied for DSM multiprocessors, where memory latencies are a major factor in limiting performance. They can be interesting also for bus-based multiprocessors, since processor speed is increasing at a much faster rate than memory. In these systems, not only pure parallel workloads, but also workloads consisting of both parallel and sequential applications are the user demand for processing power. The aim of this work is to investigate the relations among workloads of this kind, multithreaded processors, shared-bus architecture, coherence protocols, cache schemes, and thereby the consequences on performance. We have used an enhanced trace-driven approach that takes into consideration both user and kernel references. Results show that multithreaded processors can play an effective role in boosting performance even in shared-bus multiprocessors.*

Keywords – *Multithreaded architectures, Shared-bus Multiprocessors, Cache Memory, Coherence protocols.*

1. Introduction

Bus contention is major problem, in performance scaling of shared-bus multiprocessors, beside the memory access latency increase due to high processor-memory speed ratios [1, 2]. A cache contributes in both hiding memory latency and reducing the number of accesses to shared bus [3, 4]. Several techniques are used to cope with the latencies in cache-based multiprocessors: prefetching, write buffering, relaxed memory consistency models [5], and non-blocking caches [6].

Cache coherence maintaining causes overhead which weight differently due to application sharing patterns, coherence model and bus operation cost. As for the coherence protocol many solutions have been proposed in literature [7, 8, 9, 3], but when many independent processes run on our target machine, process migration becomes a source of overhead and new solutions have to be used, such as *processor affinity* [10] or the use of *passive shared copy removal* technique [11]. When a major issue is load balancing, processor affinity is not effective in all workload conditions, whilst protocol level solutions have more flexibility.

Multithreaded processors [12, 13, 14, 2] are characterized by multiple contexts that allow the processing unit to rapidly switch to execute another thread of computation whenever long latency operations occur, thus increasing processor utilization by overlapping computation with long-latency operation service. Multiple contexts have some effects on the caches: the cache miss ratio may increase, due to the breaking of locality caused by the interleaved references of different threads [4]; parallel application processes may induce a lower coherence overhead, when binding those processes to different contexts of the same processor. The global amount of shared data may increase, because the number of running processes is far larger than in the case of single-thread processors. The overall performance is a function of cache interference between multiple contexts, which therefore depends on the features of the application [15, 16], the number of contexts per processor, the context switching strategy, the cache features, and the coherence protocol.

Trace-driven simulation has been typically used to evaluate the performance of such machines [17, 18]. Some important aspects concerning accuracy [19, 20, 21] must be taken into consideration both in the trace generation and in the utilization phases. Trace Factory uses an hybrid approach in which user references were provided by a tracing tool and kernel references were synthesized from kernel patterns of actual multiprocessor traces, to achieve trace completeness [22]. To give feedback to the delays generated by the simulated architecture, Trace Factory [23] uses an *on-demand trace-driven* technique in which references are supplied to the simulator only when needed. The evaluation workloads are tailored for a Unix-like machine.

The paper is organized as follows: Section 2 presents existing solutions for multithreading and cache coherence, motivating our choices, Section 3 explains the methodology of evaluation, and Section 4 shows the main results of experiments conducted on significant actual workloads.

2. Multithreading and Cache Coherence

Multithreaded architectures have been intensively studied as a way for reducing long latencies in DSM multiprocessors, and may be interesting also for bus-based multiprocessor. Many solutions have been proposed and studied for the event(s) triggering the context switch operation [2, 24]: *switch-on-every-instruction-fetch* is limited by the great amount of contexts necessary to keep the execution pipeline full [25]; *switch-on-miss* works with a smaller degree of multithreading but causes inter-thread-conflict-misses, which can break the locality of each thread [26], and also may cause a loss of shared data that requires extra coherence overhead [15]; *instruction interleaving* performing better on a uniprocessor, but the difference with the previous scheme becomes negligible on multiprocessors [26]. Other variants are: *simultaneous multithreading* [27], *switch-on-block-of-instructions*, *switch-on-every-load* [28], and *switch-on-load-miss*. Once a switch is decided, one of the available contexts is activated and several options have been presented in literature: *thread priorization* [29], *MRU thread* [30], but generally a *round robin* is used.

Multithreading helps on latency hiding, but also bus traf-

fic reduction plays an important role. A coherence protocol aware of the application sharing pattern can reduce the number of bus operations [31]. In this paper we consider a *write-update* protocol (Dragon [7]), a *write-invalidate* protocol (Berkeley [9]) and PSCR [11] which uses a selective invalidation technique to limit the number of *passive shared copies*, i.e. the copies of private data generated by process migration. Process migration plays an important role in a general-purpose multiprocessor, since it allows the programmer to develop his applications without caring about load balance.

The interaction between multithreading and coherence has been pointed out in [24] and is therefore explored here by highlighting two new aspects: i) the kind of machine, which is the shared-bus shared-memory multiprocessor, ii) the kind of workload, which consists of a mix of both uniprocess and multiprocess applications, including kernel aspects. The shared-bus multithreaded multiprocessor has been evaluated considering a switch-on-load-miss strategy for the switching algorithm and a round-robin scheme for thread scheduling at context level and comparing some different coherence protocol schemes (Write-Update, Write-Invalidate, and selective invalidation).

3. Methodology of Evaluation

A good trade-off between cost and accuracy, for evaluating the memory subsystem, is represented by trace-driven simulation [32, 18]. Starting from a set of *source* traces including only user references we produced complete multiprocessor *target* traces [33], including also kernel references. Source traces has been obtained using TangoLite [20]. Target traces are generated by considering source traces, number of processors, number of contexts per processor, and the following three kernel activities: i) *kernel memory references*, i.e. the references due to each system call and kernel management routine; ii) *process scheduling*, i.e., the dynamic assignment of a ready process to an available thread; and iii) *virtual-to-physical address translation*, i.e., the mapping of virtual to physical memory addresses. To correctly reproduce the temporal sequence of all events, a new reference is generated whenever a request comes from the simulator (*on demand* policy). We called such technique *on-demand trace-driven* [22, 23] for the sake of differentiating it from the trace-driven technique [34].

In our approach, kernel reference bursts are modeled by gathering statistics from a series of eight-processor traces distributed by Carnegie Mellon University and obtained on an Encore Multimax (shared-bus multiprocessor) machine [35]. As for the bursts we consider their length and reciprocal distance. An evaluation of the error induced by this method has been carried out in [36].

Another aspect of kernel which influences performance is the scheduling strategy, which should provide an acceptable degree of load balance. Nevertheless, load balance induces process migration that causes *passive sharing* [17] and *process-migration sharing* [2, 8]: a memory block belonging to a private area of a process can be replicated in more than one cache as a consequence of the migration of the process which owns this block. These copies have to be treated as shared with respect to the coherence-related operations, resulting in a heavy and useless burden for the shared bus.

For these purposes, two levels of scheduling have to be considered: i) a high level scheduling, concerning the dynamic allocation of threads on the available processors, and ii) a low level scheduling, managing the context switch between threads on a single processor. The high level scheduler parameters are: the number of processes (N_{proc}), the number of processors of the target machine (N_{cpu}), the number of thread contexts supported by each processor (N_{tc}), the time slice in terms of number of references (T_{slice}), the process activation algorithm (*two-phase* or *non-blocking*), and the scheduling strategy (*random*, *round-robin* or *cache-affinity*). Each thread has its own T_{slice}, and the system assumes the presence of $N_{cpu} \times N_{ct}$ contexts. The low level scheduler uses a round-robin policy.

Class	Parameter	Timings
CPU	read cycle	2
	write cycle	2
	duration of each slice (cycles)	14
	maximum number of references per slice	2
	Probability of 0 references per slice	0.1
	Probability of 1 references per slice	0.3
	Probability of 2 references per slice	0.6
Cache	cache size	256 kbytes
	block size	64 bytes
	state updating	1
	write cycle	1
	read cycle	1
Bus	width	128 bit
	Write transaction	5
	Invalidate transaction	5
	Update-Block transaction	10
	Memory-Read-Block transaction	72
	Cache-Read-Block transaction	16

TABLE 1. NUMERICAL VALUES OF THE REFERENCE SIMULATED SYSTEM (TIMINGS ARE SPECIFIED IN TERMS OF CLOCK CYCLES).

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

TABLE 2. STATISTICS OF UNIPROCESS APPLICATION AND UNIX COMMAND TRACES (64-BYTE BLOCK SIZE, 2,500,000 REFERENCES).

4. Performance evaluation

The simulated system consists of N independent processors with 256-KByte private cache, that are interconnected to a single shared bus to access main memory. The coherence schemes Dragon [7], Berkeley [9], PSCR [11] have been used. As base case study, a machine with 128-bit shared-bus and 64-Byte cache block is considered below. As for the number of processors, four configurations have been considered (3, 6, 9 and 12) and the hardware contexts explored for each processor configuration are 1 (no multithreading), 2 and 3. The simulated processors are MIPS-R3000-like; paging relays on 4-KByte page size; scheduler time slice (T_{slice}) is 200,000 references; the execution time

Workload	Processors	Distinct blocks	Code (%)	Data		Shared blocks	Shared data (%)		Write-run			
				Read (%)	Write (%)		Accesses	Write	WRL		XRR	
									μ	σ	μ	σ
MP3D	2	5173	78.14	15.22	6.64	913	9.10	2.22	11.88	10.83	2.15	2.65
	4	6480	78.56	14.30	7.14	1625	10.34	3.20	8.18	6.04	1.54	1.60
	6	6923	78.70	13.99	7.31	2004	10.91	3.56	7.03	5.06	1.51	1.63
Cholesky	2	14312	79.35	12.88	7.77	1	0.14	0.00	2.00	0.00	2.00	0.00
	4	17119	79.83	13.57	6.60	7215	8.29	1.19	4.75	3.47	1.06	0.65
	6	19172	80.21	13.65	6.14	8789	9.67	1.32	4.46	3.12	1.06	0.68

TABLE 3. STATISTICS OF MULTIPROCESS SOURCE TRACES (64-BYTE BLOCK SIZE AND 2,500,000 REFERENCES). WRL IS THE NUMBER OF WRITE OPERATIONS TO A MEMORY BLOCK PERFORMED BY A GIVEN PROCESSOR BEFORE ANOTHER PROCESSOR ACCESS THE SAME BLOCK: THE SEQUENCE OF WRITES (EVENTUALLY INTERLEAVED BY LOADS) IS CALLED WRITE-RUN. XRR INDICATES HOW MANY READ OPERATIONS USE A BLOCK AFTER A WRITE-RUN HAS BEEN TERMINATED.

Workload	PEs	Processes	Distinct Blocks	Code (%)	Data (%)		Shared blocks	Shared data (%)		Write-run			
					Read	Write)		Accesses	Write	WRL		XRR	
										μ	σ	μ	σ
UniP	3	30	31756	78.87	13.89	7.24	7545	15.44	4.52	21.15	11.54	4.88	7.76
	6	30	52564	79.15	13.97	6.88	8164	15.84	4.75	20.67	11.01	5.12	8.23
	9	30	71245	79.66	14.23	6.11	13892	16.63	5.06	20.13	10.67	5.56	8.95
	12	30	97218	79.94	14.31	5.75	13577	17.24	5.32	20.05	9.83	6.32	9.30
Mix1	3	32	44375	79.12	14.56	6.32	8078	15.23	4.49	16.12	11.23	4.58	7.02
	6	34	63847	78.95	14.21	6.84	11964	16.05	5.23	15.72	10.78	4.11	6.94
	9	34	83671	78.23	14.34	7.43	15845	16.78	5.44	13.34	9.63	3.67	5.83
	12	36	85168	78.03	14.25	7.72	17003	17.55	5.45	12.24	9.16	3.40	5.45
Mix2	3	32	45737	79.32	14.32	6.36	8345	15.45	4.41	15.96	11.56	4.73	7.62
	6	34	68194	79.01	14.12	6.87	13456	16.03	5.12	15.66	10.84	4.45	6.89
	9	34	86123	78.56	14.01	7.43	15234	16.56	5.34	13.56	9.89	4.06	5.45
	12	36	91582	78.44	14.21	7.35	17886	17.04	5.31	12.56	10.71	3.34	6.51

TABLE 4. STATISTICS OF TARGET TRACES (64-BYTE BLOCK SIZE AND 2,500,000 REFERENCES PER CPU).

analyzed corresponds to 2,500,000 references per processor. A random selection from the ready queue with a two-phase activation algorithm has been adopted for the high level scheduler. The base case study timings and parameter values for the simulator are summarized in Table 1.

4.1. Workloads

Typical workloads for the target machine under analysis are constituted of uniprocess applications, Unix commands and multiprocess applications. A number of Unix commands (`awk`, `cp`, `du`, `lex`, `rm` and `ls`) with different command line options, some utility programs (`cjpeg`, `djpeg` and `gzip`), a network application (`telnet`) and a user application (`msim`, the multiprocessor simulator used in this work) have been selected. Traces have been taken during different execution sections of the application: the initial (*beg*) and middle (*mid*) sections. Table 2 describes the features of these source traces in terms of number of distinct blocks used by the program code, data read and write access percentages, and number of system calls.

The two parallel programs used in the case studies, `mp3d` and `Cholesky`, are well-known programs from the SPLASH suite. Table 3 summarizes the statistics concerning multiprocess application traces; it also specifies the number of shared blocks and some statistics that characterize the access pattern to shared blocks such as the *write-run length* (*WRL*) and *external re-reads* (*XRR*) [17].

The performance evaluation has been conducted using three workloads: *UniP* has been set up selecting 30 uniprocess applications; *Mix1* and *Mix2* consist of 30 uniprocess applications and an additional load due to a parallel application (`MP3D` and `Cholesky`, respectively), that generates a number of processes equal to half the total number of processors available on the machine.

Table 4 reports target traces statistics. The simulation which produced these traces was performed with a 256-KByte, direct-mapped cache with 64-Byte block size and the Dragon protocol. Comparing the write-run statistics in Tables 3 and 4, we can notice how the presence of kernel activities and uniprocess applications modifies the write-run of source traces, in particular because of process migration.

4.2. Simulation results

Let us consider workload *UniP*, that consists of independent sequential processes only. Bus utilization ratio has not a clear trend for all processor configurations (Figure 1).

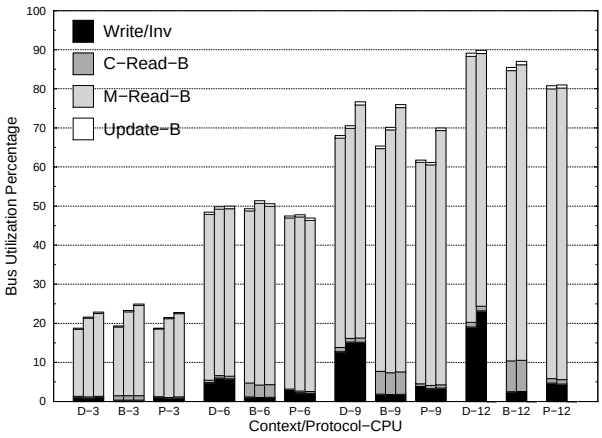

FIGURE 1. BUS UTILIZATION RATIO FOR DIRECT-ACCES CACHES AND UNIP WORKLOAD VS. NUMBER OF PROCESSORS (3, 6, 9, 12), TYPE OF COHERENCE PROTOCOL (DRAGON, BERKELEY, PSCR), AND NUMBER OF CONTEXTS (1, 2, 3). FOR EACH PROTOCOL-CPU CONFIGURATION (D, B, P INDICATE DRAGON, BERKELEY, AND PSCR, RESPECTIVELY), THREE CONTEXT CONFIGURATIONS ARE SHOWN. IN THE CASE OF 12 PROCESSORS ONLY TWO CONFIGURATIONS HAVE BEEN CONSIDERED DUE TO THE INSUFFICIENT NUMBER OF PROCESSES TO ACHIEVE ONE PROCESS FOR EACH CONTEXT. FOR EVERY CONFIGURATION BUS UTILIZATION PERCENTAGE OF WRITES OR INVALIDATIONS FOR BERKELEY (WRITE/INV), CACHE-READ-BLOCKS (C-READ-B), MEMORY-READ-BLOCKS (M-READ-B), UPDATE-BLOCKS (UPDATE-B) ARE SHOWN.

Anyway, we can observe that:

- the bus load generally increases with the number of contexts (the reason for this variation is mainly due to Read-Block transactions);
- Dragon protocol exhibits a greater utilization of the bus in respect with the other protocols, whilst PSCR exhibits a lower one; this trend appears more clearly for configurations with a higher number of processors;

- Write transactions[1]are almost constant with the number of contexts, although they increase with the number of processors (i.e. number of caches). Only Dragon protocol has an increase of Write transactions.

Taking in mind that we are considering a workload entirely constituted of sequential programs, the increase of Write transactions, with the number of processors, can be reconducted to the increase of shared copies due to passive sharing and kernel.

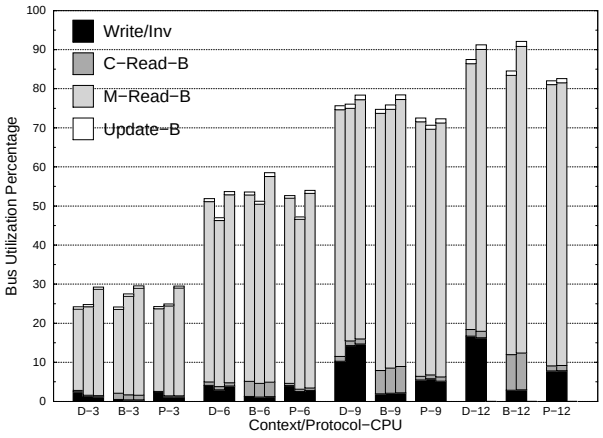

FIGURE 2. BUS UTILIZATION FOR DIRECT-ACCESS CACHE AND MIX1 WORKLOAD.

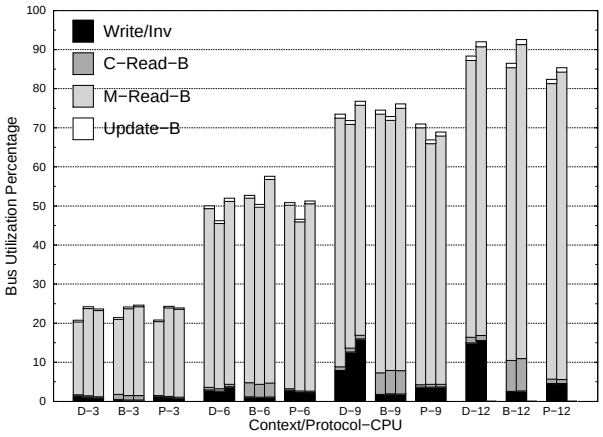

FIGURE 3. BUS UTILIZATION FOR DIRECT-ACCESS CACHE AND MIX2 WORKLOAD.

When a parallel application is added to the previous workload (*Mix1*) the bus utilization ratio increases in respect with the previous reference case. This is due to higher number of Write (Invalidations) and Read-Block transactions (due to invalidations) necessary to keep coherent the greater number of shared copies generated by the parallel application itself (Figure 2). Moreover, higher number of processes in *Mix1* causes a miss increase.

Again, in this case the bus utilization increase is not constant for all processor configurations; an additional reason is that, in some cases, processes belonging to the parallel application are executed onto the same processor. Thus, binding more parallel processes on the same processor while diminishing parallelism, has the benefit of reducing the bus load. This result seems in contrast with [37], but in that study workloads were constituted of parallel applications only, while here sequential programs are considered too.

Thus, the designer may accept a tradeoff between having an increased scalability or a greater speed for the parallel application. This tradeoff may have a difficult solution since a general purpose multiprocessor makes use of the concept of process priority. While this concept is not necessary in a parallel machine, in a general purpose machine the operating system can assign a lower priority to parallel application, and give a single program a higher priority or vice-versa.

Also the workload *Mix2* confirms the same considerations just said for *Mix1*. In both cases there is not a great influence by Write-run, instead the coherence protocol is of a major importance (Figure 3).

The higher miss rate trend with the number of contexts, observed in the previous cases and shown in detail in Table 5 for *UniP* workload, suggests to analyze also caches with higher associativity, while maintaining the same cache size (256 KBytes).

Let us take into consideration again the *UniP* workload with a 4-way cache, the bus utilization ratio decreases (Figure 4) with respect to the same direct-access configuration. This is due to the better cache utilization by the running threads, i.e. the different localities take noticeable advantage from having more cache associativity to exploit.

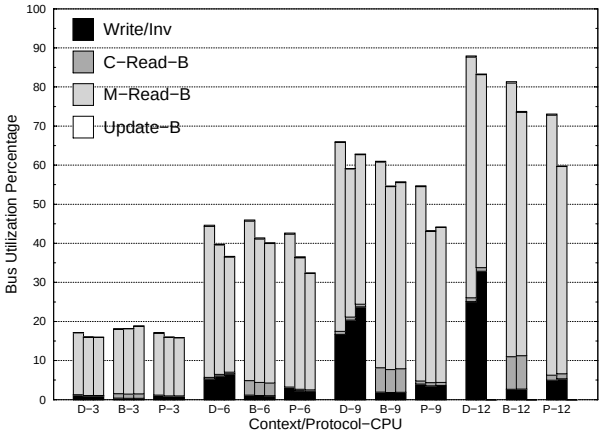

FIGURE 4. BUS UTILIZATION FOR A 4-WAY CACHE AND UNIP WORKLOAD.

A 4-way cache allows a longer average life of cache blocks: this causes a lower miss ratio (and a lower number of Read-Block transactions) but also an higher number of shared copies, and thus an higher number of Write transactions. The increase of Writes is strongly dependent on the type of coherence protocol (Table 5 and 6).

Moving the attention to the analysis of the *Mix1* workload we observe that the miss ratio gain over direct-access case is less evident (Figure 5).

5. Conclusions

In this paper, we have considered multithreaded shared-bus multiprocessors. For these machines new kind of workloads have been analyzed: those that include also uniprocess applications. Such workloads can cause passive shar-

[1]For Berkeley protocol, Invalidation transactions are showed instead of Write transactions, since the protocol provides this method to keep coherence.

Parameter	Processors	Dragon			Berkeley			PSCR		
		1	2	3	1	2	3	1	2	3
Bus Util. Ratio (%)	3	18.732	21.588	22.872	19.293	23.265	24.945	18.758	21.467	22.739
	6	48.428	49.818	49.990	49.283	51.370	50.586	47.440	47.757	46.940
	9	68.041	70.526	76.660	65.355	70.149	75.965	61.739	61.129	70.010
	12	89.100	89.824	—	85.466	87.025	—	80.782	80.986	—
Miss Rate (%)	3	0.401	0.464	0.490	0.503	0.587	0.625	0.400	0.460	0.489
	6	0.533	0.526	0.524	0.714	0.702	0.689	0.530	0.531	0.517
	9	0.478	0.472	0.548	0.683	0.708	0.777	0.477	0.468	0.555
	12	0.522	0.502	—	0.737	0.741	—	0.506	0.509	—
Write/Inv. Trans. per 100 CPU Operations	3	0.358	0.299	0.339	0.113	0.111	0.112	0.331	0.267	0.309
	6	0.812	0.998	0.957	0.194	0.168	0.170	0.476	0.366	0.339
	9	1.503	1.782	1.851	0.219	0.206	0.216	0.439	0.367	0.395
	12	1.948	2.388	—	0.242	0.243	—	0.435	0.406	—

TABLE 5. BUS UTILIZATION PERCENTAGE, MISS RATE AND WRITES/INVALIDATIONS FOR DIRECT-ACCES CACHE AND UNIP WORKLOAD VS. NUMBER OF CPUS (3 ,6,9,12), TYPE OF COHERENCE PROTOCOL (DRAGON, BERKELEY, PSCR), AND NUMBER OF CONTEXTS (1, 2, 3). MISS RATE IS ALSO THE NUMBER OF READ-BLOCK BUS TRANSACTIONS PER 100 PROCESSOR OPERATIONS, AS WELL AS WRITE/INV IS THE NUMBER OF WRITES (OR INVALIDATIONS FOR BERKELEY) BUS TRANSACTIONS PER 100 PROCESSOR OPERATIONS.

Parameter	Processors	Dragon			Berkeley			PSCR		
		1	2	3	1	2	3	1	2	3
Bus Util. Ratio (%)	3	17.181	16.029	15.972	18.111	18.207	18.829	17.100	16.009	15.894
	6	44.612	39.793	36.627	45.961	41.366	40.170	42.616	36.554	32.436
	9	65.994	59.145	62.817	60.985	54.672	55.745	54.685	43.206	44.186
	12	87.914	83.324	—	81.341	73.734	—	73.084	59.820	—
Miss Rate (%)	3	0.360	0.334	0.333	0.483	0.476	0.490	0.360	0.336	0.333
	6	0.469	0.391	0.346	0.675	0.584	0.566	0.467	0.391	0.345
	9	0.418	0.318	0.321	0.648	0.565	0.584	0.413	0.315	0.323
	12	0.462	0.348	—	0.700	0.632	—	0.442	0.345	—
Write/Inv. Trans. per 100 CPU Operations	3	0.367	0.300	0.297	0.116	0.108	0.111	0.327	0.262	0.263
	6	0.876	0.973	1.053	0.196	0.170	0.163	0.475	0.363	0.335
	9	1.931	2.260	2.650	0.226	0.204	0.212	0.448	0.378	0.388
	12	2.530	3.071	—	0.246	0.243	—	0.440	0.448	—

TABLE 6. BUS UTILIZATION PERCENTAGE, MISS RATE AND WRITES/INVALIDATIONS FOR 4-WAY CACHE AND UNIP WORKLOAD.

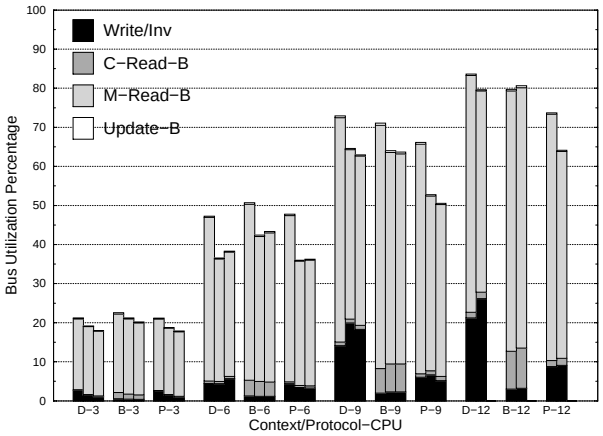

FIGURE 5. BUS UTILIZATION FOR A 4-WAY CACHE WITH MIX1 WORKLOAD.

ing, and a coherence protocol aware of that (for example PSCR), can achieve better performance. We found that multithreaded processors help to increase performance by reducing the bus load and thus allowing more scalability, but more cache associativity should be used to better exploit multithreading. Moreover, having multiple contexts and cache associativity, the sharing pattern exhibited by an additional parallel application running concurrently with the sequential programs becomes less important. Anyway, in all cases the coherence protocol can make the difference. Further studies will explore the tradeoffs of distributing the available parallelism among processors and threads.

6. Acknowledgments

This work was supported by Ministero della Università e della Ricerca Scientifica e Tecnologica (MURST), Italy. Thanks to Steve Herrod at Stanford University for providing and helping with TangoLite. The traces distributed by Carnegie Mellon University, were collected by Bart Vashaw with the assistance and supervision of Drew Wilson of Encore Computer Corporation and Dan Siewiorek of Carnegie Mellon University.

References

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

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

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

[4] D. Lenoski and W.-D. Weber, *Scalable shared-memory multiprocessing*. Morgan Kaufmann Publishers, San Francisco, CA, 1995.

[5] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29 (12), pp. 66–76, December 1996.

[6] G. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, (New York, NY), pp. 53–62, ACM Press, April 1991.

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

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

[9] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 276–283, June 1985.

[10] M. S. Squillante and D. E. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *IEEE Transaction on Parallel and Distribuited Systems*, vol. 4 (2), pp. 131–143, February 1993.

[11] C. A. Prete, G. Prina, and L. Ricciardi, "A selective invalidation strategy for cache coherence," *IEICE Transaction on Information and Systems*, vol. E78-D (10), pp. 1316–1320, October 1995.

[12] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz, "April: a processor architecture for multiprocessing," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 104–114, May 1990.

[13] R. Saavedra-Barrera, D. Culler, and T. von Eicken, "Analysis of multithreaded architectures for parallel computing," in *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pp. 169–178, July 1990.

[14] J. Laudon, A. Gupta, and M. Horowitz, "Architectural and implementation tradeoffs in the design of multiple-context processors," in *Multithreaded Computer Architecture: A Summary of the State of the Art* (R. A. Iannucci, G. R. Gao, R. Halstead, and B. Smith, eds.), pp. 167–200, Kluwer Academic Publisher, 1994.

[15] R. Thekkath and S. J. Eggers, "The effectiveness of multiple hardware contexts," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 328–337, October 1994.

[16] W.-D. Weber and A. Gupta, "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 273–280, June 1989.

[17] S. J. Eggers, "Simulation analysis of data sharing in shared memory multiprocessors," PhD thesis UCB/CSD 89/501, Univ. of California, Berkeley, April 1989.

[18] C. A. Prete, G. Prina, and L. Ricciardi, "A trace-driven simulator for performance evaluation of cache-based multiprocessor system," *IEEE Transaction on Parallel and Distributed Systems*, vol. 6 (9), pp. 915–929, September 1995.

[19] M. A. Hollyday and C. S. Ellis, "Accuracy of memory reference traces of parallel computations in trace-driven simulations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 1, pp. 97–109, January 1992.

[20] S. R. Goldschmidt, *Software Coherence in Multiprocessor Memory Systems.* PhD thesis, Stanford University, Computer Systems Laboratory, June 1993.

[21] C. B. Stunkel, B. Janssens, and W. K. Fuchs, "Address tracing of parallel systems via trapeds," *Microprocessors and Microsystems*, vol. 16, no. 5, pp. 249–261, 1992.

[22] R. Giorgi, C. Prete, G. Prina, and L. Ricciardi, "A hybrid approach to trace generation for performance evaluation of shared-bus multiprocessors," in *Proceedings 22nd EuroMicro International Conference, Prague*, pp. 207–214, September 1996.

[23] R. Giorgi, C. Prete, G. Prina, and L. Ricciardi, "Trace factory: a workload generation environment for trace-driven simulation of shared-bus multiprocessor," *to appear on IEEE Concurrency*.

[24] A. Agarwal, "Performance tradeoffs in multithreaded processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3 (5), pp. 525–539, September 1992.

[25] D. M. Tullsen and S. J. Eggers, "Effective cache prefetching on bus-based multiprocessors," *ACM Transactions on Computer Systems*, vol. 13, no. 1, pp. 57–88, February 1995.

[26] J. Laudon, A. Gupta, and M. Horowitz, "Interleaving: A multithreading technique targeting multiprocessors and workstations," in *Proceedings oh the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 308–318, Oct. 1994.

[27] D. M. Tullsen, S. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pp. 392–403, June 1995.

[28] H. L. Muller, P. W. A. Stallard, and D. H. D. Warren, "Multitasking and multithreading on a multiprocessor with virtual shared memory," in *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*, pp. 212–221, February 1996.

[29] J. James and A. S. Fiske, "Thread scheduling mechanisms for multiple-context parallel processors," Tech. Rep. AITR-1545, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts, June 1995.

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

[31] A. Gupta and W.-D. Weber, "Cache invalidation patterns in shared-memory multiprocessors," *IEEE Transactions on Computers*, vol. C-41 (7), pp. 794–810, July 1992.

[32] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherency protocols," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, (Jerusalem, Israel), pp. 2–15, May 28–June 1, 1989.

[33] 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.

[34] S. R. Goldschmidt and J. L. Hennessy, "The accuracy of trace-driven simulation of multiprocessors," in *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (B. D. Gaither, ed.), pp. 146–157, May 1993.

[35] B. Vashaw, "Address trace collection for trace driven simulation of bus based, shared memory multiprocessors," technical report CMUCDS-93-4, Carnegie Mellon University, Pittsburgh, PA, March 1993.

[36] R. Giorgi, C. Prete, G. Prina, and L. Ricciardi, "A workload generation environment for trace-driven simulation of shared-bus multiprocessor," in *Proceedings 30th HICSS, Hawaii, IEEE Press*, pp. 266–275, January 1997.

[37] R. Thekkath and S. J. Eggers, "Impact of sharing-based thread placement on multithreaded architecture," in *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pp. 176–186, April 1994.