A Workload Generation Environment for Trace-Driven Simulation of Shared-Bus Multiprocessors

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

Dipartimento di Ingegneria dell'Informazione
Facoltà di Ingegneria, Università di Pisa
Via Diotisalvi, 2 - 56126 PISA (Italy)
prete@iet.unipi.it

Abstract

We describe an environment to produce traces representing significant workloads for a shared-bus shared-memory multiprocessor used as a general-purpose multitasking machine, where each processor can include multithread facilities. By means of an exclusively software approach, the environment produces traces that include both user and kernel references, starting from source traces containing only user references. The process scheduling and the virtual-to-physical address translation are simulated, whereas a stochastic model is provided for the generation of the kernel reference stream. The paper includes a Section describing the generation of three different workloads used to evaluate the performance of a shared-bus shared-memory multiprocessor.

1. Introduction

A shared-bus shared-memory multiprocessor architecture represents a low-cost solution for high performance general-purpose workstations, where the major concern is to speed-up the execution of a set of system commands, uniprocess applications and/or multiprocess applications with coarse/medium grain parallelism [1]. This architecture can be considered a simple extension of a uniprocessor machine [2]; both kinds of machines generally use the same operating system model (a Unix-like multitasking processing environment is the most common solution) and the same user application can be executed on these machines without recoding.

An intrinsic limitation of the shared-bus architecture is the low number of processors which can be connected to the shared bus before saturation (about a few dozen units is the upper limit allowed by current technology) [1, 2]. Cache memories and multithreading are two of the solutions proposed in literature to limit the drop of performance due to bus saturation. The cache [3, 4] contributes to a reduction in the number of accesses to the shared bus, while the multithreading reduces the impact of long-latency operations [5, 6]. In a multiprocessor architecture with private caches, the coherence-related overhead becomes a major cause of the drop of global performance. When two or more processors store a copy of the same memory block in their respective caches and one of them performs a write operation on a location within that block, some bus actions (induced by a coherence protocol [7, 8, 9, 10, 11]) are performed to guarantee that each subsequent read operation by any processor may get the up-to-date value of the modified location.

Multithreaded processors [12, 13, 5, 2] include multiple contexts that allow the processing unit to switch to another thread when the current thread is forced to wait for a memory operation. The multiple contexts cause a double effect: on the one hand, the cache miss ratio increases, due to the breaking of locality caused by the interleaved references of different threads [4]; on the other hand, if a number of processes belonging to a parallel application are considered as threads concurrently running on a single processor, their impact on bus bandwidth due to coherence overhead is significantly lower than it would be if they were scheduled on different processors. However, the global amount of shared data tends to 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 due to multiple contexts, therefore it strictly depends on the features of the application [6], the number of threads, the context switching strategy, and the coherence protocol adopted.

Trace-driven simulation is typically used in order to in-

In Proc. 30th HICSS, Hawaii, Jan 1997, pp.266-275

vestigate how to improve performance of such machines, because this method has the advantage of not being strictly linked to a specific kind of architecture and being, therefore, flexible in the performance evaluation of different architectures [14]. A key point of this methodology is to find address traces which both represent typical operating conditions and include all information potentially needed for a reliable simulation of the system. Some important aspects concerning accuracy [15, 16, 17] must be taken into consideration both in the trace generation phase and in the utilization phase. When recording traces, the most critical issue is trace completeness, i.e., traces must include both user and operating system address references. Furthermore, a minimum amount of time distortion must be introduced by the tracing mechanism. In the utilization phase, one must be sure to reproduce the correct synchronization between events concerning processes run on different processors.

In the present work we introduce a methodology and a set of tools (called *Trace Factory*) to generate traces for performance evaluation of a shared-memory multiprocessor system (e.g. multiprocessor workstation architectures), where each processor may have a multithreaded architecture. For this purpose a set of typical Unix-like workloads may be generated by: i) tracing a set of uniprocess applications (e.g. commonly used Unix commands and user programs) and/or multiprocess applications, and then ii) adding the kernel activities which most influence global performance, such as process scheduling, virtual-to-physical address translation, and reference stream generated by the kernel routines. Both process scheduling and memory mapping are simulated, whereas the kernel reference stream is modeled by means of a statistic method already proposed and validated by the authors in [18]. In the case of a multithreaded architecture, Trace Factory considers multiple threads for each processor, and the simulator implements a proper scheduling policy at the thread level.

The paper is organized as follows: Section 2 presents a brief overview of the open literature on the subject; Section 3 describes the general organization of the tracing environment; Section 4 focuses on the methodology employed for kernel modeling; Section 5 shows an example of trace generation with three different workloads for performance evaluation of a multiprocessor; finally, Section 6 concludes the paper.

2. Related work

A number of different strategies have been employed in the literature concerning performance evaluation of multiprocessor systems: analytical/stochastic models, trace-driven simulation, complete system simulation, just to mention the most common solutions.

The analytical/stochastic model [19] appears to be the most flexible and economic solution, yet the low degree of accuracy which it provides may be unacceptable in the case of cache performance evaluation, since the model does not typically include all aspects which characterize cache and program behavior (e.g. coherence in multiprocessor systems). This kind of analysis could be of some use to obtain a quick estimation of system performance, before performing the actual evaluation by means of a more reliable technique.

The methodology based on complete simulation (such as SIMICS [20], SimOS [21], or MINT [22]) is the most flexible and accurate, since it allows a detailed analysis of all hardware and software aspects and cause-effect links involved in a particular architecture. On the other hand, this solution involves high implementation costs to obtain an acceptable degree of accuracy. The main limitation of this solution is that the slowdown scales linearly with the number of CPUs being simulated, since the detailed CPU model does not exploit the parallelism of the underlying machine. This causes a slowdown factor in the thousands when simulating systems with 16–32 processors with the deepest level of simulation detail being required [21].

When the target of the performance evaluation is the memory subsystem, a good trade-off between cost and accuracy is represented by trace-driven simulation [23], which we already experimented with successfully in [11, 18]. Trace-driven methods are based on the recording of a trace (sequence of memory addresses referenced by the running program) and on the utilization of the trace as input for the memory subsystem simulator. Two critical issues concerning accuracy in tracing shared-memory multiprocessors are: i) traces must include both user and operating system references, and ii) a minimal amount of time distortion must be induced either by the tracing mechanism (during the recording phase) or by the simulator (in the utilization phase).

Tracing techniques include hardware and software solutions. Hardware monitoring [24] is the solution which can potentially guarantee best results in terms of accuracy. Indeed, actual traces, captured by traditional hardware-based techniques, prove to be particularly useful in the validation phase of a new architecture. Accuracy, absence of time distortion and of intrusiveness are the main advantages of this method. The most critical drawback comes from the fact that modern trends in technology for processors encourage the adoption of on-chip caches, so that a great amount of memory references are handled internally and can no longer be captured by the hardware tracing mechanism [25]. Other limiting factors are the high cost of implementation and the lack of completeness (fragmentation) of the trace gathered, due to the limited size of storage buffers. Finally, traces obtained from an actual multiprocessor machine by means of hardware techniques cannot be employed for an exhaustive performance analysis of the system, because it is not possible to produce traces with a variable number of processors,

and the designer of a new multiprocessor cannot generally access the trace system to produce new traces.

Software tracing methodologies include *program instrumentation*, *single-step execution* and *microcode modification*. A major problem, which at various levels affects all the methods just mentioned, is *time dilation*, due to the fact that the software tracing mechanisms generate a heavy overhead, which causes major changes in the relative timing of asynchronous events, resulting in a decrease of accuracy, especially concerning multitasking workloads.

Program instrumentation (e.g. TangoLite [16], MPTRACE [26] and TRAPEDS [17]) is based on the modification of program code to record the sequence of memory accesses generated during the execution. A set of instructions are added to create the trace section concerning each *basic block* (set of machine-level instructions which are always executed in sequence without interruptions) throughout the program. The instrumentation phase may be activated at either source or executable level. The latter is the simplest to handle for the user, but it is quite difficult to implement and may not be able to instrument all programs; on the other hand, instrumentation at source level is quite easy, but the user must have access to the entire source code and have a full knowledge of the compiler's code generation process. Furthermore, the accuracy of the model is limited by the lack of completeness in the trace, since it is quite difficult to capture traces of operating system routines.

The problem of completeness also arises in single-step execution. This technique can be adopted with microprocessors which allow the execution of a program to be interrupted after each instruction. Nevertheless, since operating system routines typically disable interruptions, no possibility usually exists to capture addresses generated by the execution of operating system routines.

The tracing technique based on microcode modification (ATUM) uses processor microcode to record addresses in a reserved part of main memory as a side effect of normal execution [27]. Compared with other techniques, this one leads to fewer distortions and a very fast recording (only 10x slowdown); all the system activity can be observed, with no additional hardware being required. The disadvantages include poor flexibility, since it requires microcode modification. This also implies that only microcode activity can be recorded, whereas I/O operations and hardware-generated memory traffic cannot be captured.

In any case, when the goal is to compare different architecture solutions, it becomes important to analyze the system behavior under a predefined and controlled workload. Two key points of this approach are: i) traces must represent actual workloads for the target machine, and ii) the designer must have the possibility to produce proper traces to investigate the behavior of the system when exposed to particular (possibly critical) workload conditions. This kind of flexibility can be guaranteed only by software techniques, and this is the main reason that we directed our efforts in this direction in the present work. A possible solution to the issues exposed above is presented, so that the traces generated appear to be suitable for a thorough investigation of a given target architecture.

3. The Trace Factory environment

Trace Factory allows the utilization of a set of *source* traces including only user references to produce complete multiprocessor *target* traces. Source traces can be obtained by a tool [25] based on the same microprocessor as in the target system. Target traces are generated by considering the source traces, the number of processors, the number of threads for each 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 processor; 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 in the system, the production of the target traces is made according to the *on demand* policy (a new reference is generated whenever a request comes from the simulator), and the scheduler makes use of the synchronization tags inserted into the trace files by the tracing mechanism. The interaction between Trace Factory and the simulator is synchronous, so that the trace generated is forced to follow the temporal behavior imposed by the memory subsystem.

3.1. Generation of kernel references

Kernel reference bursts affect performance because they interrupt the locality of the memory reference stream of the running process causing additional cache misses. In our approach, kernel bursts are obtained by inserting sequences of kernel references within the user reference stream. These sequences are generated by means of two statistics: *length* of each burst and *distance* between the starting point of two subsequent bursts. The burst insertion may also be driven by information collected in the source traces if the tracing tool records the system call positions. This allows us to generate more accurate workloads in the case of processes that exhibit different numbers of system calls.

Each kernel reference is specified by: *area referenced* (code/data), *address* within the selected area and *kind of access* (read/write). The probability of code/data access and of data read/write access are input parameters for the tool. For the specific location within the selected area, the locality of memory references has to be taken into account; the

stochastic model used in Trace Factory is introduced in [18]. The reference generator operates as follows: let R_i be the address of the latest reference in an area (R_0 assumes a random value). The address R_{i+1} may be evaluated through a sequence of steps: first, we evaluate the relative distance (in words) $y = |R_{i+1} - R_i|$ by transforming a uniformly-distributed random variable $x \in (0, 1)$ by means of the (empiric) function

$$y = \left\lfloor \frac{S \cdot (1 - A^x)}{5 \cdot A^x - 6} \right\rfloor \quad (1)$$

where A and S express the locality of references in the area involved and are jointly related to the width and amplitude peak of the curve representing the distribution of distances between subsequent references (see Section 4.1).

Once we have evaluated y in such a way, we transform it into $\tilde{y}$ by changing its sign with probability p_b that represents the probability of backward references. Finally, if the access being considered involves the code area and the value of $\tilde{y}$ is positive, this value is incremented by one, in order to prevent zero distance between two subsequent accesses in code area. The resulting value of $\tilde{y}$ is added to R_i and the outcome is assumed to be the required address R_{i+1}.

3.2. Process management

In the case of shared-bus shared-memory multiprocessors, one of the main goals of the scheduler is to provide an acceptable degree of load balance in order to allow the programmer to develop his applications without caring about the workload distribution on the processors. Nevertheless, load balance induces process migration that causes further coherence overhead. Actually, 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 (*passive sharing* [14], *process-migration sharing* [2]). A scheduling policy based on cache affinity [28] can reduce the amount of cache misses due to context switches and passive sharing.

Trace Factory simulates a simple scheduler by considering the following input parameters: the number of processes (N_{proc}), the number of processors of the target machine (N_{cpu}), the number of threads supported by each processor (N_{thr}), 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*). It starts from a set of source traces (one trace for each uniprocess application and as many traces as the number of processes belonging to the multiprocess application), and produces as many target traces as the number of processors of the target machine.

The scheduler operates as follows: if a process *p* is running on processor *P* for a *D* time interval (again specified in terms of number of references) then *D* references of the *p* source trace become references for processor *P*. At the start of the simulation, all the processes are ready and they are inserted in a proper queue, namely R_1. Initially, the scheduler randomly selects N_{cpu} processes, and each running process has a different time slice (namely, the process running on processor i is assigned a time slice $T_i = \frac{i \cdot T_{slice}}{N_{cpu}}$). After the first context switch on each processor the next scheduled process is regularly assigned T_{slice}. This strategy, typically adopted in operating systems for multiprocessors, avoids a context switch being simultaneously needed on each processor every T_{slice}, which would produce an undesirable overlap of miss peaks on all caches and a consequent bus saturation due to the bus transactions needed to fetch missing blocks.

On a context switch, a process is extracted from R_1 and assigned to the available processor. The choice of such process can be made either according to the cache affinity strategy, or by means of a round-robin scheme, or just randomly. The preempted process may be managed in two different ways. In the *non-blocking* activation strategy, the preempted process is immediately inserted into the R_1 queue. This strategy suffers from the starvation problem: this implies that references of a process may be not present within a target trace when its length is short and $\frac{N_{proc}}{N_{cpu}} \gg 1$. A second activation strategy (*two-phase*) makes use of another queue, namely R_2, initially empty. On every context switch, the preempted process is inserted into R_2 (phase one). As soon as the queue R_1 becomes empty, all the processes are taken from R_2 and inserted into R_1 (phase two). This technique avoids the problem described above, that is, it ensures that a process does not have to wait an indefinite time for its turn: indeed, with this strategy, a process cannot be executed $n + 1$ times before each other process is executed exactly n times.

Finally, the scheduler can consider the synchronization sequence produced by a multiprocess application execution. In this case, the process scheduling is driven by the time slice for processes belonging to uniprocess applications and by both the time slice and the synchronization sequence for multiprocess applications. Source traces have to include synchronization tags for a correct playback of them [24]. When a process reaches an out-of-order synchronization event, it is inserted into a waiting queue to wait for the synchronization event. Then, it enters either the R_1 or the R_2 queue as described above.

In the case of multithreaded architecture, each thread is assigned its own T_{slice}, and the system assumes the presence of $N_{cpu} \times N_{thr}$ virtual processors. Two levels of

scheduling need to be considered: i) an external scheduling (described above), concerning the dynamic allocation of threads on the available processors in the system, and ii) and internal scheduling, concerning the context switching between threads on a single processor. The internal level of scheduling is completely handled by the simulator and it is a simple round-robin policy.

The number of threads supported by each processor, the cost of context switch between threads, and the context switch strategy are the parameters which the simulator has to be supplied with. In particular, three different context-switch policies are considered: *switch on miss*, *switch on read miss* and *switch on block of instructions*. These techniques involve different cache behaviors with respect in particular to the resulting miss rate and the overhead induced by the coherence protocol [2].

3.3. Virtual-to-physical address translation

In virtual memory models based on paging, the localities of virtual and physical references produced by a running process may be different. The mapping of sequential virtual pages of the program into non-sequential physical pages causes this difference and influences the number of *intrinsic interference* (or *capacity*) *misses* due to interferences among kernel code and data, user data and code accesses within the same cache set.

The virtual-to-physical address translation is modeled as follows. We suppose that each process has a private address space and a shared address space common to all the processes belonging to the same multiprocess application, whereas, kernel instances share a unique address space. The virtualization is implemented using a paging scheme *on demand* and *without prepaging*. The page size and the physical memory size are input parameters for the address translation mechanism.

4. The kernel behavior modeling

4.1. Parameter gathering

The stochastic model of kernel references is described by a set of parameters that can be gathered from an actual trace including kernel references. The probabilities concerning the kind of area referenced and the access mode (read/write) can be evaluated by counting the relative occurrences of events.

Locality information (that is, A, S and p_b) has to be extracted from the traces, so that the reference stream generated by Eq. 1 is an accurate representation of the actual stream. The following data are separately evaluated for code and data areas in the actual traces: i) the maximum distance (Δ) between two subsequent references; ii) the maximum amplitude (P) of the distribution of distances between two subsequent references; iii) percentage of backward references (p_b) over the total number of non-sequential accesses. To determine a numerical value for the A and S parameters, we derive the following conditions:

Application	Source	Brief description
ecas	A. Wilson (Encore)	Computer Architecture Simulation
hartstone	N. Weiderman (SEI at CMU)	Real-Time Benchmark
locusroute	SPLASH (by J. Rose)	Circuit Routing
mp3d	SPLASH (by J.D. McDonald)	Rarefied Fluid Flow Simulation
ms_tracer	Fritzz Graphics (M. Rao at CMU)	Ray Tracing
pde	A. Wilson	Partial Differential Equation Solver

Table 1. The CMU multiprocessor traces

1. the maximum value for the y distribution (Eq. 1.) is set to the maximum amplitude (P) of the distribution of distances between two subsequent references; this leads to the equation:

$$P = \frac{1}{\log A} \log \frac{S+6}{S+5} ; \tag{2}$$

2. the maximum value of y (Eq. 1.) is set to the maximum distance (Δ) between two subsequent references; this leads to the equation:

$$\Delta = S \frac{1-A}{5A-6} . \tag{3}$$

Eqs. 2 and 3 may be linked together to build a system which can be easily solved with numerical techniques.

Finally, we measure the distribution of the kernel burst length and of the distance between the beginning of two successive bursts.

In our case study, we gathered the kernel statistics from a series of eight-processor traces distributed by Carnegie Mellon University and obtained by means of an Encore Multimax (shared-bus multiprocessor) machine (Table 1). These traces represent a wide variety of application domains [24]. They include system references and each reference involves one of four different areas: user code, user data, kernel code and kernel data. For the sake of simplicity, we analyzed a sample of 1,250,000 references per processor for each trace. Table 2 includes the kernel access percentages (code, data, write), the kernel burst statistics and the resulting values of A, S and p_b. The statistics concerning the distribution of distance and burst length are summarized by means of average value (μ) and standard deviation (σ).

4.2. Validation of the model

To estimate the error induced by the synthetic generation of the kernel reference stream, trace-driven simulation

Application	Kernel references (%)	Kernel burst distance μ	Kernel burst distance σ	Kernel burst length μ	Kernel burst length σ
ecas	3.45	28098	583	974	440
hartstone	13.13	13877	8718	583	1198
locusroute	7.39	23474	5234	1213	1486
mp3d	3.39	28316	1100	966	290
ms_tracer	9.42	22382	5502	1007	2406
pde	5.63	24046	7078	967	1000

Application	Kernel code: References (%)	A	S	p_h	Kernel data: References (%)	Writes (%)	A	S	p_h
ecas	2.08	1.198834	0.772	0.357	1.37	0.47	1.197679	19.900	0.727
hartstone	8.09	1.199707	0.810	0.356	5.04	1.46	1.198834	21.086	0.698
locusroute	4.10	1.198834	0.827	0.363	3.29	1.37	1.197679	20.459	0.729
mp3d	2.04	1.198834	0.210	0.372	1.35	0.46	1.197679	19.900	0.727
ms_tracer	5.98	1.199854	0.832	0.332	3.44	0.84	1.199707	22.032	0.732
pde	3.47	1.199707	0.783	0.309	2.16	0.80	1.199707	21.837	0.738

Table 2. Kernel references statistics (8-CPU multiprocessor)

	Application	(i)	(ii) abs	(ii) err (%)	(iii) abs	(iii) err (%)	(iv) abs	(iv) err (%)	(v) abs	(v) err (%)
Miss rate	ecas	0.273	0.225	−17.5	0.283	+3.6	0.284	+4.0	0.290	+6.2
	hartstone	0.071	0.016	−77.0	0.081	+14.0	0.075	+5.6	0.088	+23.9
	locusroute	0.292	0.113	−61.3	0.295	+1.0	0.267	−8.6	0.280	−4.1
	mp3d	0.522	0.473	−9.4	0.528	+1.1	0.528	+1.1	0.530	+1.5
	ms_tracer	0.071	0.017	−76.0	0.099	+39.4	0.097	+36.6	0.096	+35.2
	pde	0.10	0.06	−40.0	0.12	+20.0	0.12	+20.0	0.13	+30.0
Global System Power	ecas	684.8	709.6	+3.6	672.3	−1.8	672.2	−1.8	666.8	−2.6
	hartstone	780.4	796.7	+2.1	781.4	+0.1	785.9	+0.7	781.8	+0.2
	locusroute	723.8	776.3	+7.2	717.9	−0.8	722.0	−0.2	713.7	−1.4
	mp3d	632.1	662.4	+4.8	630.9	−0.1	628.3	−0.6	627.4	−0.7
	ms_tracer	763.5	755.3	−1.1	772.0	+1.1	757.9	−0.7	752.4	−1.4
	pde	780.5	792.0	+1.5	780.8	+0.0	779.5	−0.1	777.0	−0.4
Write transactions per 1000 memory operations	ecas	47.5	50.5	+6.3	49.7	+4.6	49.3	+3.7	50.0	+5.2
	hartstone	10.58	3.47	−67.2	8.55	−19.2	7.56	−28.5	8.78	−17.0
	locusroute	8.44	6.76	−19.9	7.29	−13.6	7.16	−15.1	7.50	−11.1
	mp3d	0.88	0.13	−85.2	1.09	+23.8	1.09	+23.8	1.21	+37.5
	ms_tracer	46.2	67.5	+46.1	42.0	−9.1	48.0	+3.9	49.1	+6.3
	pde	13.2	9.6	−27.2	12.2	−7.5	11.3	−14.4	12.9	−2.2

Table 3. Kernel model validation (8-CPU multiprocessor, Dragon protocol)

was employed to compare the results of five different situations: i) the original CMU traces; ii) the CMU traces deprived of the kernel references; iii) the original traces, with kernel references replaced by an address stream generated synthetically, yet preserving the same position and length of each kernel reference burst as in the original traces; iv) as in iii), with kernel bursts length and position generated stochastically by means of a distribution evaluated on a per-processor basis from the original traces; v) as in iv), with the distributions evaluated as average values over the entire set of processors.

Since the kernel references represent only a minimal part of total accesses, the analysis was extended to cover 2,500,000 references in order to get an amount of data significant enough to be used for statistic considerations.

The typical metrics representing the performance of a multiprocessor system were considered: average miss rate, Global System Power ($GSP = \sum U_{CPU}$, where $U_{CPU}=\frac{T_{cpu}-T_{delay}}{T_{cpu}} \times 100$ and T_{delay} is the total CPU delay time spent in waiting for memory operation completions), bus utilization ratio, number of write/read-block/update-block transactions per thousand memory operations.

Table 3 shows some results in terms of both absolute values and error percentages (with respect to the values obtained from the actual traces). Cases iv) and v), in particular, point out that it is not generally worth reproducing exactly the original position of kernel accesses within the global reference stream in order to improve the accuracy of the model.

5. Performance evaluation of a multiprocessor

A possible, quite interesting use of the Trace Factory environment is to perform the analysis of a shared-bus shared-memory multiprocessor employed as a high performance general-purpose machine (e.g. as a Unix workstation). The hardware system consists of N independent processors; each processor has a private cache, and the processors are connected to a single shared bus to access the main memory.

5.1. Workload selection

Typical workloads for the target machine considered in this example are: i) a set of uniprocess applications and Unix commands and ii) a mixed set of uniprocess applications, Unix commands and multiprocess applications. In the latter case, we considered two typical sub-cases: coarse- and medium-grain parallel applications. We selected a number of typical Unix commands (`awk`, `cp`, `du`, `lex`, `rm` and `ls`) with different command line options, some utility programs (`cjpeg`, `djpeg` and `gzip`), a network application (`telnet`) and a user application (`msim`, the multiprocessor simulator used in this work). Traces are taken during different execution sections of the application: the initial (*beg*) and middle (*mid*) sections. Table 4 describes the features of these source traces in terms of number of distinct (unique) blocks used by the program, code, data read and data write access percentages, and number of system calls.

The two parallel programs used in the case studies, `mp3d` and `Cholesky`, come from the SPLASH suite; in both applications, one of the input parameters is the number of processes on which the computation is parallelized. The `mp3d` program simulates rarefied hypersonic flow; the trace generated is relative to the case of 10,000 molecules and 20 time steps. `Cholesky` performs the factorization of a sparse positive definite matrix using the homonymous method; the trace was generated using, as input, a 1806-by-1806 matrix with 30,284 non-zeros elements coming from the Boing/Harwell sparse matrix test (`bcsttk14`). The source traces are produced by means of the TangoLite tool; the parallel application is traced on a virtual multiprocessor having as many processors as the number of application processes. Table 5 summarizes the statistics concerning multiprocess application traces; it also specifies the number

Application	Distinct blocks	Code (%)	Data (%) read	Data (%) write	System calls
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
djpeg (mid)	144157	98.33	1.17	0.51	20
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 4. Statistics of uniprocess application and Unix command traces (64-byte block size, 1,250,000 references)

Workload	Processors	Shared blocks	Shared data (%) Accesses	Shared data (%) Write	Write-run WRL μ	Write-run WRL σ	Write-run XRR μ	Write-run XRR σ
mp3d	2	1335	9.10	2.29	6.96	7.14	1.95	2.38
	4	2137	9.97	3.12	5.97	4.29	1.54	1.29
	6	2450	10.30	3.39	5.27	3.41	1.51	1.23
	8	2742	10.52	3.54	4.96	3.10	1.51	1.27
	10	2983	10.69	3.64	4.78	2.91	1.51	1.32
	12	3172	10.81	3.71	4.71	2.91	1.51	1.37
	14	3312	10.89	3.75	4.61	2.86	1.51	1.42
Cholesky	2	1	0.16	0.00	2.00	0.00	2.00	0.00
	4	7806	7.10	0.96	2.78	1.62	1.06	0.65
	6	12415	9.23	1.36	2.81	1.54	1.04	0.59
	8	14278	10.14	1.46	2.78	1.50	1.04	0.56
	10	15527	10.64	1.51	2.77	1.43	1.04	0.63
	12	16888	10.78	1.50	2.85	1.51	1.04	0.51
	14	17937	10.89	1.48	2.91	1.58	1.04	0.55

Table 5. Statistics of multiprocess source traces (64-byte block size, 1,250,000 references)

of shared blocks and some statistics concerning the access pattern to shared blocks.

Access patterns to shared data influence the multiprocessor performance and may be characterized by means of two metrics: *write-run length* (*WRL*) and *external re-reads* (*XRR*) [14]. The former is the number of write operations to a memory block performed by a given processor before another processor would access the same block: the sequence of write (eventually interleaved by read) references, just defined, is called a *write-run*. The second one indicates how many read operations will use a block after a write-run has been terminated and before another one has been initiated.

5.2. The target trace production

Traces are generated choosing the following details concerning the underlying architecture of the simulated multiprocessor: first of all, processors are MIPS-R3000-like and their number (N_{cpu}) has been varied from 2 to 24; the paging has been carried out using a page size of 4 KBytes; the time slice (T_{slice}) is 200,000 references; the execution time analyzed corresponds to 1,250,000 references per processor. A random selection from the ready queue with a two-phase activation algorithm was adopted for the choice of a process on context switches.

Workloads represent a typical situation in which various users run some typical UNIX commands and different ordinary applications in different times. For this reason, workloads include two or three copies of the same program taken in different execution sections (*beg* and *mid*). In particular, workload *UniP* has been set up selecting 31 uniprocess applications from Table 1; workloads *Mix1* and *Mix2* consist of 30 uniprocess applications and an additional load due to a parallel applications which generates a number of processes equal to half the total number of processors available on the machine.

Table 6 describes the features of target traces. 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 5 and 6, 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. This aspect strongly motivates the introduction of kernel modeling in the analysis of such architectures.

Workload	Processors	Shared blocks	Shared data (%) Accesses	Shared data (%) Write	Write-run WRL μ	Write-run WRL σ	Write-run XRR μ	Write-run XRR σ
UniP	8	6843	13.73	4.14	5.97	8.06	3.27	5.17
	12	12914	16.42	5.00	6.44	8.36	3.42	5.31
	16	17557	17.40	5.30	6.67	8.52	3.89	5.94
	20	21247	17.82	5.50	6.68	8.53	4.01	6.13
	24	23358	17.88	5.52	6.78	8.58	4.04	6.17
Mix1	8	7772	12.56	3.86	6.25	7.29	2.88	4.70
	12	14153	17.16	5.28	5.97	6.89	2.75	4.33
	16	22711	18.09	5.78	5.65	6.55	2.62	3.61
	20	29199	18.44	5.88	5.33	5.86	2.80	3.71
	24	31434	18.84	6.07	5.14	5.56	2.72	3.60
Mix2	8	8226	12.14	3.58	5.51	7.23	3.08	5.03
	12	16103	16.94	5.24	5.37	7.26	2.84	4.74
	16	23018	17.51	5.43	5.18	7.23	2.36	4.09
	20	30189	17.68	5.46	4.94	6.79	2.47	4.37
	24	33406	18.08	5.55	4.80	6.63	2.38	4.31

Table 6. Statistics of target traces (64-byte block size, 1,250,000 references per CPU)

5.3. Simulation results

In the simulator, a multiprocessor is described in terms of CPU, cache and bus parameters. The CPU features are described by the clock cycle, the minimal number of clock cycles for a read/write operation and the temporal distribution of the memory accesses. To describe that distribution we define the *slice* as a fixed lapse of time corresponding to a constant sequence of CPU clock cycles. The parameters which specify the distribution are: the length of the slice, the maximum number of memory references per slice (M) and the probability that in the slice there are exactly

Class	Parameter	Timing I	Timing II
CPU	read cycle	2	2
	write cycle	2	2
	average duration of each slice (cycles)	14	14
	maximum number of references per slice	2	2
	Probability of 0 references per slice	0.1	0.1
	Probability of 1 references per slice	0.3	0.3
	Probability of 2 references per slice	0.6	0.6
Cache	cache size	256 kbytes	256 Kbytes
	block size	64 bytes	128 bytes
	associativity	1	1
	state updating	1	1
	write cycle	1	1
	read cycle	1	1
Bus	width	64 bit	64 bit
	write transaction	5	5
	update-block transaction	18	34
	read-block transaction	32	48
	cache-to-cache read-block transaction	22	38

Table 7. Numerical values of some input parameters for the multiprocessor simulator (times are specified in terms of clock cycles)

$0, 1, 2, \ldots, M$ memory references. Traditional parameters describe the cache structure. Finally, the bus is described by the number of CPU clock cycles for each kind of transaction: write, update-block, memory read-block, and cache-to-cache read-block.

The target architecture of our analysis is a shared-bus multiprocessor with a 256-Kbyte, direct-mapped cache; the bus width is assumed to be 64 bits. Two block sizes are considered: 64 bytes and 128 bytes, which involve two different set of timings for bus transactions (I and II in Table 7, respectively). The following coherence protocols are considered: Dragon [7], Update-Once [8], Berkeley [9], Competitive Snoopy Caching [10] and PSCR [11]. The Dragon protocol is based on a *write-update* strategy, whereas the other protocols use an invalidation strategy to limit the coherence overhead. In particular, PSCR employs a selective invalidation technique in order to limit the number of *passive shared copies*: it invalidates the copies belonging to private data area of a process as soon as they are fetched by another processor. Berkeley and Update-Once invalidate on the first or on the second write on a shared copy, respectively. Competitive Snoopy Caching switches from *write-update* to *write-invalidate* for each cached block when the number of cycles for write broadcasts issued equals the sum of the cycles potentially needed to reread the block. This technique limits the coherence-related overhead to twice the optimal value.

In the case of the *UniP* workload, the actual shared areas belong only to the kernel, and the process migration creates a high number of useless *passive shared copies*. Since PSCR systematically destroys these passive shared copies, it provides better performances than other protocols (Figure 1). For the same reason all the protocols that invalidate shared copies have better performance than Dragon. Since the various protocols adopt different invalidation strategies, the difference between timing costs for write and read-block

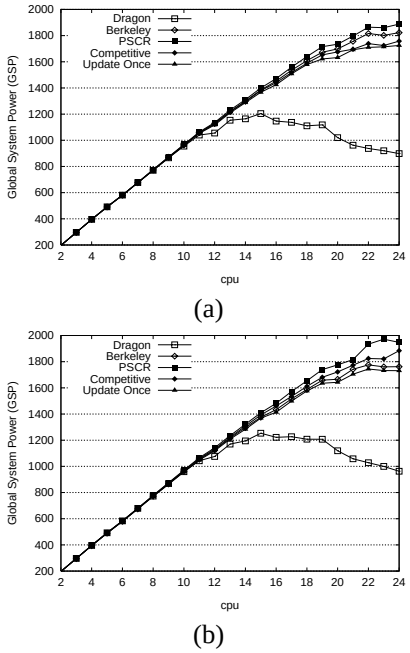

Figure 1. Global System Power of multiprocessor executing the UniP workload. (a) Timing I. (b) Timing II.

transactions causes a different reciprocal behavior of the protocols. In particular, PSCR and Competitive Snoopy Caching take advantage of the situation labeled as 'Timing II' (128-byte block-size).

Figure 2 shows the system behavior with the *Mix1* workload. The introduction of a coarse-grain parallel application determines a drop in global performance for each protocol, due to the coherence-overhead introduced by the actual shared copies. Even in this case, the behavior changes with the block size and Berkeley is the protocol which exhibits the worst penalization with 128-byte block size.

Finally, Figure 3 shows that a medium-grain parallel application (*Mix2* workload) introduces a lower overhead.

6. Conclusions and further research

We have shown an environment which permits us to produce traces including kernel and user references starting from traces containing only user references. The target system is a shared-bus shared-memory multiprocessor, where each processor can either have a traditional architecture or include multithread facilities. Specifically, the kernel reference sequences, which appear in actual traces as bursts which break the program locality, have been produced by means of a stochastical model. This model, in turn, has been calibrated on data extracted from actual traces. In the ex-

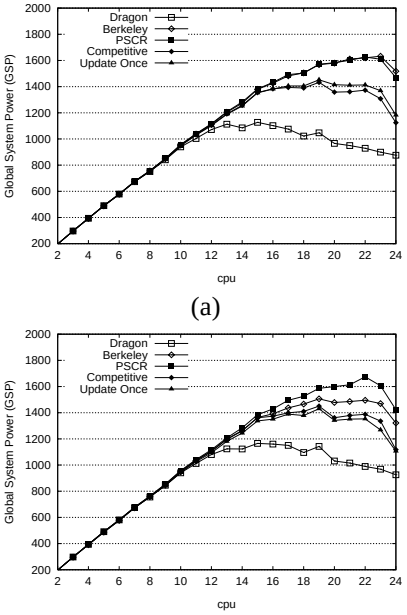

Figure 2. Global System Power of multiprocessor executing the Mix1 workload. (a) Timing I. (b) Timing II.

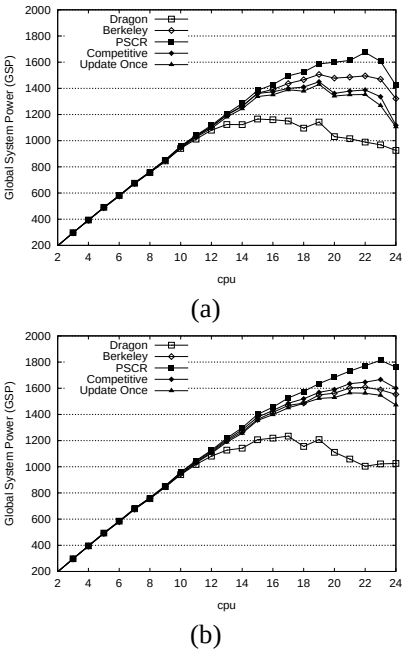

Figure 3. Global System Power of multiprocessor executing the Mix2 workload. (a) Timing I. (b) Timing II.

amples, the methodology used to gather those data has been shown and validated for a trace set furnished by Carnegie Mellon University. Furthermore, Trace Factory models the scheduling and the translation from virtual to physical addresses. Both activities have a notable impact on the behavior of a system including caches: process migration, normally produced by a scheduler, produces *passive sharing* and a consequent large amount of bus overhead to guarantee the coherence of shared copies; the remapping of addresses influences the number of *intrinsic interference misses* due to code and data accesses of a given process involving the same cache set.

The future goals of our research activity include the generation of workloads suitable to evaluate the impact of multithreading on the efficiency of different cache structures and coherence protocols.

7. Acknowledgments

This work was supported by the 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 multiprocessor 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. We are also grateful to Pierfrancesco Foglia for his contribution to the validation of the methodology. Finally, we thank Veljko Milutinović for his valuable comments and suggestions.

References

[1] J. Hennessy, and D. A. Patterson, *Computer Architecture – a Quantitative Approach*, 2nd edition, Morgan Kaufmann, 1996.

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

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

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

[5] J. Laudon, A. Gupta, and M. Horowitz, "Architectural and implementation trade-offs in the design of multiple-context processors", Stanford University Technical Report CSL-TR-92-523, May 1992.

[6] W.D. Weber and A. Gupta, “Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results”, in *Proc. 16th. Int. Symp. Comput. Arch.*, June 1989, pp. 273-280.

[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] J.D. Gee, and A.J. Smith, “Evaluation of cache consistency algorithm performance”, *Proc. Int’l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (Mascots)*, San Jose, CA, February 1996, pp. 236-249.

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

[10] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator, “Competitive Snoopy Caching”, *Proceedings of the 27th Ann. Symp. on Foundations of Computer Science*, 1986, pp. 244-254.

[11] C.A. Prete, G. Prina, and L. Ricciardi, “A selective invalidation strategy for cache coherence”, *IEICE Trans. Information and Systems*, vol. E78-D, n. 10, October 1995, pp. 1316-20.

[12] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz, “April: a processor architecture for multiprocessing”, in *Proc. 17th. Int. Symp. Comput. Arch.*, May 1990, pp. 104-114.

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

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

[15] M.A. Holliday and C.S. Ellis, “Accuracy of memory reference traces of parallel computations in trace-driven simulation”, *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, n. 1, January 1992, pp. 97-109.

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

[17] C.B. Stunkel, B. Janssens, and W.K. Fuchs, “Address tracing of parallel systems via TRAPEDS”, *Microprocessors and Microsyst.*, vol. 16 n. 5, 1992, pp. 249-261.

[18] C.A. Prete, G. Prina and L. Ricciardi, “A trace-driven simulator for performance evaluation of cache-based multiprocessor systems”, *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, n. 9, September 1995, pp. 915-29.

[19] M.K. Vernon, E.D. Lazowska, and J. Zahorian, “An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols”, in *Proc. 15th Int. Symp. Comput. Arch.*, pp. 308-315, May 1988.

[20] P. Magnusson and B. Werner, “Efficient memory simulation in SIMICS”, Technical Report, Swedish Institute of Computer Science, 1995.

[21] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta, “Complete computer simulation: the SimOS approach”, *IEEE Parallel and Distributed Technology*, vol. 3, n. 4, Winter 1995, pp. 34-43.

[22] J.E. Veenstra and R.J. Fowler, “MINT: a front end for efficient simulation of shared-memory multiprocessors”, *Proc. 2nd Int.’l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (Mascots)*, Durham, NC, January 1994, pp. 201-207.

[23] S. Eggers and R.H. Katz, “Evaluating the performance of four snooping cache coherency protocols”, *Proc. 16th Int. Symp. Comput. Arch.*, 1989, pp. 2-15.

[24] B. Vashaw, “Address trace collection and trace-driven simulation of bus based, shared memory multiprocessors”, Research Report, Dept. of Elec. and Comp. Eng., Carnegie Mellon Univ., Pittsburgh, PA, March 1993.

[25] C.B. Stunkel, B. Janssens, W.K. Fuchs, “Address tracing for parallel machines”, *IEEE Computer*, vol. 24, n. 1, Jan. 1991, pp. 31-38.

[26] S. Eggers, D. Keppel, E. Koldinger and H. Levy, “Techniques for efficient inline tracing on a shared-memory multiprocessor”, *Proc. ACM SIGMetrics Int. Conf. Measurement and Modeling of Computer Systems*, 1990, pp. 37-47.

[27] R.L. Sites and A. Agarwal, “Multiprocessor cache analysis using ATUM”, *Proc. 15th Int. Symp. Comput. Arch.*, 1988, pp. 186-195.

[28] M.S. Squillante and D.E. Lazowska, “Using processor-cache affinity information in shared-memory multiprocessor scheduling”, *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, n. 2, pp. 131-143, Feb. 1993.