A Hybrid Approach to Trace Generation for Performance Evaluation of Shared-Bus Multiprocessors

Roberto Giorgi
Cosimo Antonio Prete
Luigi Ricciardi

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

Gianpaolo Prina

Scuola Superiore di Studi Universitari
e di Perfezionamento Sant'Anna
Via Carducci, 40 - 56100 PISA (Italy)
gianpaolo@sssup1.sssup.it

Abstract

This paper describes a hybrid methodology (based on both actual and synthetic reference streams) to produce traces representing significant complete workloads. By means of a software approach, we generate traces that include both user and kernel references, starting from source traces containing only user references. We consider the aspects of kernel that have a deeper impact on the multiprocessor performance by i) simulating the process scheduling and the virtual-to-physical address translation, and ii) stochastically modeling the kernel reference stream. The target system of our study is a shared-bus shared-memory multiprocessor used as a general-purpose machine with a multitasking operating system.

1. Introduction

A shared-bus shared-memory multiprocessor 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 commands, uniprocess applications and/or multiprocess applications with coarse/medium grain parallelism [7]. However, an intrinsic limitation of the shared-bus architecture is the low number of processors which can be connected to the shared bus; when this number exceeds a critical value (about a few dozen units is the upper limit allowed by current technology), the system undergoes a drastic drop in global performance, due to bus saturation [7, 9]. The adoption of private caches reduces the number of accesses to the shared bus but induces the coherence problem [5, 10, 11, 13, 15, 21] which is a major source of overhead.

A number of different strategies have been employed in the literature concerning performance evaluation of multiprocessor systems: analytical/stochastic models [24], trace-driven simulation, complete system simulation [12, 16, 23], just to mention the most common solutions. 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 [3]; 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 [1].

Trace-driven methods are based on the generation of a trace (sequence of memory addresses referenced by the running program) and on the utilization of the trace in performance evaluation of a specific architecture or set of architectures. Two critical issues concerning accuracy [6, 8, 20] 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. Accuracy, absence of time distortion and of intrusiveness are the main advantages of *hardware monitoring*. The most critical drawback of this approach 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 [19]. Furthermore, 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. Indeed, actual traces,

captured by traditional hardware-based techniques, prove to be particularly useful in the validation phase of a new architecture.

Software tracing methodologies include *program instrumentation* [4, 6, 20], *single-step execution* [2] and *microcode modification*. 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 [17]. 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 microcode modification requires access to on-chip ROM.

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 (like program instrumentation and single-step execution), and this is the main reason that we directed our efforts in this direction in the present work. However, the accuracy of these techniques is limited by the lack of completeness in the trace, since it is quite difficult to capture traces of operating system routines. A possible solution to this issue is presented, so that the traces generated appear to be suitable for a thorough investigation of a given target architecture.

In the present work we introduce a methodology and a set of tools to generate traces for performance evaluation of a shared-memory multiprocessor system (e.g. multiprocessor workstation architectures). 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 were simulated within the tool, whereas the kernel reference stream was modeled by means of a statistic method already proposed and validated by the authors in [14].

To correctly reproduce the temporal sequence of all events in the system, the production of the scheduled 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 interface between the proposed tool and the simulator is obtained by means of synchronous channels, so that the trace generated is forced to follow the temporal behavior imposed by the memory subsystem simulator. For the sake of simplicity, the term "trace" will be used in the remainder of the paper to indicate the sequence of references flowing through such channels.

2. The methodology

We use a set of *source* traces including only user references to produce complete multiprocessor *target* traces. Source traces can be obtained by a tool based on the same microprocessor used in the target system. (For example, TangoLite [6] may be used to study a MIPS-based workstation.) Target traces are generated by considering the source traces, the target machine configuration (e.g. the number of processors) and the following three kernel activities: i) *kernel memory references*, i.e., the reference bursts 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 addresses, produced by a running process, to physical memory addresses. The reference sequences can be simply stored into target trace files or supplied to the simulator via synchronous channels; in the latter case, the target trace generation is performed on the basis of the *on demand* policy: a new reference is produced when requested by the simulator, so that the trace generated is conditioned by the temporal behavior imposed by the simulation of the memory subsystem.

2.1. 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, the kernel reference stream is obtained by means of a stochastic model of *addresses*, *burst length* and *burst distance*.

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 points 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 (e.g., to consider the fact that the processes typically exhibit a different number 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 is introduced and validated in [14]. 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

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

where A and S express the locality of references in the area involved.

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}.

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 have to be separately evaluated for code and data areas: i) the maximum distance (Δ) between two subsequent references; ii) the maximum amplitude (P) of the distribution of distances between two subsequent references; iii) the percentage of backward references (p_b) over the total number of non-sequential accesses. The Δ and P values measured on the actual traces are used to evaluate the proper values for parameters A and S. In order to determine a numerical value for these parameters, we derive the following conditions:

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 (e.g. setting S to an arbitrarily large value and substituting recursively into the equations until the process reaches convergence).

Finally, we measure the distribution of the kernel burst length and of the distance between the beginning of two successive bursts. These distributions are input data to synthetically generate the kernel reference stream.

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 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).

Application	Kernel references (%)	Kernel burst distance μ	distance σ	length μ	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_b	Kernel data References (%)	Writes (%)	A	S	p_b
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 1. Kernel references statistics (CMU trace set, 1,250,000 references per CPU, 8-CPU multiprocessor)

2.2. Process management

One of the main goals of the multiprocessor 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* [1], *process-migration sharing* [9]). Furthermore, on every context switch, a burst of cache misses occurs, due to the loading of the working set of the new process. A scheduling policy based on cache affinity [18] can reduce the amount of cache misses due to this fact.

Our tool models the process management aspects by simulating a simple scheduler. The input parameters for the scheduler are: the number of processes (N_{proc}), the number of processors of the target machine (N_{cpu}), the time slice in terms of number of references (T_{slice}) and the process activation algorithm (*two-phase* or *non-blocking*). The number of distinct processes to be scheduled is constant for a specific target trace. The tool simulates the scheduler in the following way: i) 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 ii) produces as many target traces as the number of processors of the target machine. The whole scheduling activity can be directly driven by the simulator; in this case, the scheduler operates in connection with the simulator via synchronous channels, and the scheduling activity is conditioned by the speed of each simulated processor.

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 from memory.

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 mentioned above, 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 (Figure 1). 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.

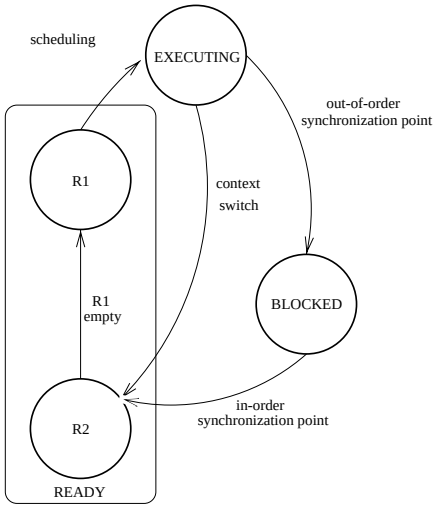

Figure 1. State transition diagram in the case of two-phase activation strategy.

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; those tags are sequential numbers representin the actual synchronization sequence of the parallel application execution [22]. When a process reaches an out-of-order synchronization event (corresponding to a tag in the trace), 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.

2.3. 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 o *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.

3. An example of target trace generation

A possible, quite interesting use of the the proposed methodology 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).

A typical workload for the target machine considered in this example is a mixed set of uniprocess applications, Unix commands and multiprocess applications. Since the performance of a multiprocessor also depends on the grain size of parallel applications, we considered two typical subcases: 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). In some cases, traces are 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 (unique) blocks used by the program, code, data read and data write access percentages, and number of system calls.

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
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 2. Statistics of uniprocess applications and Unix command traces (32-byte block size, 1,250,000 references)

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 MIPS-based multiprocessor having as many processors as the number of application processes. Virtual processors are then emulated on a single-processor MIPS workstation executing sequentially the resulting multiprogrammed load. Table 3 summarizes the statistics concerning multiprocess application traces; it also specifies the number of shared blocks and some statistics concerning the access pattern to shared blocks.

Application	CPUs	Distinct blocks	Code (%)	Data (%)		Shared blocks
				read	write	
mp3d	2	9727	78.03	15.01	6.97	1335
	4	12588	78.48	14.22	7.31	2137
	6	13532	78.65	13.95	7.41	2450
	8	14040	78.72	13.81	7.46	2742
	10	14347	78.77	13.74	7.49	2983
	12	14550	78.80	13.68	7.51	3172
	14	14742	78.82	13.65	7.52	3312
Cholesky	2	17102	79.05	11.97	8.98	1
	4	24011	79.54	13.21	7.26	7806
	6	26154	79.77	13.51	6.71	12415
	8	28564	79.95	13.66	6.38	14278
	10	30980	80.21	13.69	6.10	15527
	12	33493	80.38	13.68	5.94	16888
	14	35424	80.44	13.69	5.86	17937

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

Table 3. Statistics of multiprocess source traces (32-byte block size, 1,250,000 references)

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*) [1]. 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 de-

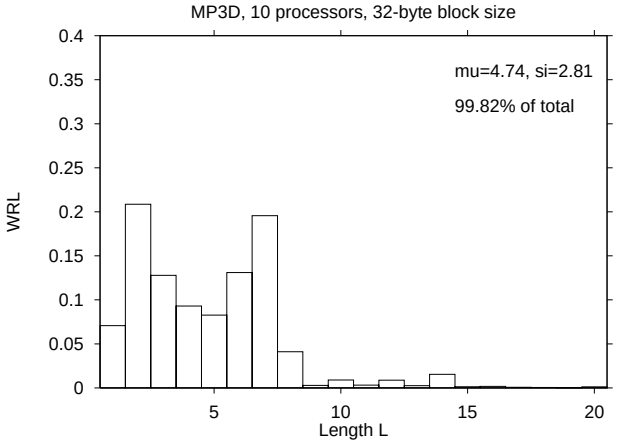

Figure 2. Global write-run-length (WRL) distribution for the mp3d workload.

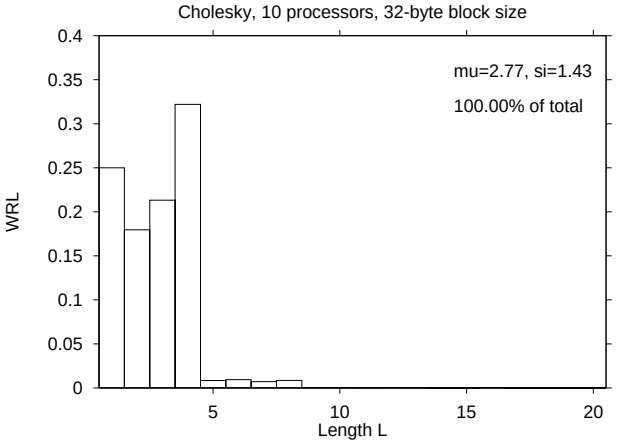

Figure 3. Global write-run-length (WRL) distribution for the Cholesky workload.

fined, 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. The tool can extract these statistics, which are shown in Table 3. A quite natural use of the write-run is to select the better coherence strategy between *write-invalidate* and *write-update* for a given workload. Long write-runs suggest that a write-invalidate coherence protocol should be chosen: indeed, the cost of the initial miss (due to invalidation) is balanced by the fact that a large amount of bus traffic can be saved since all the subsequent write operations can be executed locally during the write-run. A large number of external rereads gives an indication on how much a copy is needed by different processors and, therefore, if it would be convenient to adopt a write-update strategy. The write-run length is also a measure of the application grain-size; in fact these statistics show that: i) `mp3d` is a coarse-grained application since the average value of write-run length varies in the range ($4.61 \div 6.96$); ii) `Cholesky` exhibits a medium-grained behavior having an average value write-run length in the range ($2.00 \div 2.91$). Figures 2 and 3 stress the differences between the shared data access patterns of `mp3d` and `Cholesky`.

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.

In a typical situation, a number of users run some UNIX commands and different ordinary applications in different times. For this reason, the proposed workloads include two or three copies of the same program taken in different execution sections (*beg* and *mid*). In particular, 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.

Workload	CPUs	Distinct blocks	Code (%)	Data (%)		Shared blocks
				read	write	
Mix1	8	78218	79.05	14.15	6.80	7772
	12	94381	78.80	14.42	6.79	14153
	16	100941	78.49	14.43	7.09	22711
	20	132713	78.74	14.17	7.08	29199
	24	159569	78.82	14.11	7.07	31434
Mix2	8	79713	79.23	14.16	6.61	8226
	12	100057	79.02	14.26	6.72	16103
	16	108540	78.74	14.44	6.82	23018
	20	142314	79.03	14.19	6.78	30189
	24	170678	79.18	14.16	6.66	33406

Workload	CPUs	Shared data (%)		Write-run			
		Accesses	Write	WRL		XRR	
				μ	σ	μ	σ
Mix1	8	12.56	3.86	6.25	7.29	2.88	4.70
	12	17.16	5.28	5.97	6.89	2.75	4.33
	16	18.09	5.78	5.65	6.55	2.62	3.61
	20	18.44	5.88	5.33	5.86	2.80	3.71
	24	18.84	6.07	5.14	5.56	2.72	3.60
Mix2	8	12.14	3.58	5.51	7.23	3.08	5.03
	12	16.94	5.24	5.37	7.26	2.84	4.74
	16	17.51	5.43	5.18	7.23	2.36	4.09
	20	17.68	5.46	4.94	6.79	2.47	4.37
	24	18.08	5.55	4.80	6.63	2.38	4.31

Table 4. Statistics of target traces (32-byte block size, 1,250,000 references per CPU)

Table 4 describes the features of target traces. The simulation which produced these traces was performed with a 256-Kbyte, 2-way set-associative cache with 32-byte block size and the Dragon protocol. The difference between the *Mix1* and *Mix2* workloads is in the grain size of the paral-

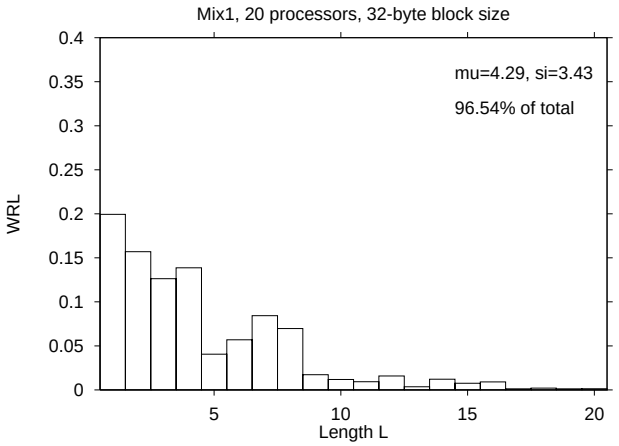

Figure 4. Global write-run-length (WRL) distribution for the Mix1 workload.

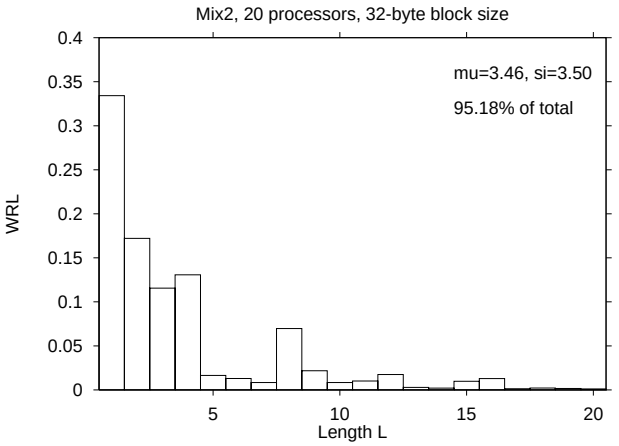

Figure 5. Global write-run-length (WRL) distribution for the Mix2 workload.

lel application. In the first case (*Mix1*), this application is `mp3d`, whilst in the second case (*Mix2*) it is `Cholesky`. Observing the statistics of data read and write, it emerges that these values are quite constant even if the analogous source applications values variate in a more extended range (see Table 2 and 3); indeed, there is a kind of averaging effect in putting together a number of applications in order to generate a multiprocessor workload. Comparing the write-run statistics of Tables 3 and 4, we can also notice how the presence of kernel activities and uniprocess applications (in particular process migration) modifies the write-run of source traces. This aspect has been highlighted by Figures 4 and 5, which show the WRL distributions for the two workloads; therefore, it strongly motivates the introduction of kernel modeling in the evaluation of such multiprocessors.

4. Conclusions

We have shown a methodology which permits us to produce traces including kernel and user references starting from traces containing only user references. Specifically, the kernel reference sequences, which appear in actual traces as bursts which break the program locality, have been produced by means of a stochastic model. This model, in turn, has been calibrated on data extracted from actual traces. In the examples, the methodology used to gather those data has been shown for an actual trace set furnished by Carnegie Mellon University. Furthermore, the proposed tool 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, causes *passive sharing* and a consequent large amount of bus overhead to guarantee the coherency 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.

Finally, we have shown, as an example, how the methodology can be employed to individuate critical workloads for a multiprocessor.

5. 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. Bart Vashaw was supported in part by a fellowship from ONR and in part by a fellowship from Encore Computer Corporation. Finally, we thank Veljko Milutinović for his valuable comments and suggestions.

References

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

[2] S. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. *Proc. 15th Int. Symp. Comput. Architecture*, 1988, pp. 373-382.

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

[4] 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.

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

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

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

[8] M.A. Holliday and C.S. Ellis. Accuracy of memory reference traces of parallel computations in trace-driven simulation. *IEEE Trans. Parallel Distributed Syst.*, vol. 3, no. 1, January 1992, pp. 97-109.

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

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

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

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

[13] E. M. McCreight. *The Dragon computer system: an early overview*. NATO Advanced Study Institute on Microarchitecture of VLSI Computer, Urbino, Italy, July 1984.

[14] C.A. Prete, G. Prina and L. Ricciardi. A trace-driven simulator for performance evaluation of cache-based multiprocessor systems. *IEEE Trans. Parallel and Distributed Syst.*, vol. 6, no. 9, September 1995, pp. 915-29.

[15] C.A. Prete, G. Prina and L. Ricciardi. A selective invalidation strategy for cache coherence. *IEICE Trans. on Information and Systems.* vol. E78-D, no. 10, October 1995, pp. 1316-20.

[16] M. Rosenblum, S.A. Herrod, E. Witchel and A. Gupta. Complete computer simulation: the SimOS approach. *IEEE Parallel and Distributed Technology*, vol. 3, no. 4, Winter 1995, pp. 34-43.

[17] R.L. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. *Proc. 15th Int. Symp. Comput. Architecture*, 1988, p. 186-195.

[18] M.S. Squillante and D.E. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel Distributed Syst.*, vol. 4, no. 2, pp. 131-143, Feb. 1993.

[19] C.B. Stunkel, B. Janssens and W.K. Fuchs. Address tracing for parallel machines. *IEEE Computer*, vol. 24, no. 1, Jan. 1991, pp. 31-38.

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

[21] 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.

[22] 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.

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

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