Trace Factory

Generating Workloads for Trace-Driven Simulation of Shared-Bus Multiprocessors

Roberto Giorgi, Cosimo Antonio Prete, Gianpaolo Prina, and Luigi Ricciardi
University of Pisa

Trace Factory produces traces representing significant real workloads consisting of a flexible set of commands and uniprocess and multiprocess user applications. The authors evaluate its accuracy and show how it can be used to evaluate and compare the performance of five coherence protocols.

A major concern with high-performance general-purpose workstations is to speed up the execution of commands, uniprocess applications, and multiprocess applications with coarse- to medium-grain parallelism. To that end, a simple extension of a uniprocessor machine such as a shared-bus, shared-memory architecture can be employed.[1] 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 application can execute on these machines without recoding.

However, an intrinsic limitation of the shared-bus architecture is the low number of processors that can be connected to the shared bus. When this number exceeds a critical value (the upper limit for current technology is approximately a few dozen units), the system's global performance drops drastically because of bus saturation. A major cause of this drop is coherence-related overhead. 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 in that block, a set of bus actions (induced by a coherence protocol[2]) is necessary to guarantee that every subsequent read operation by any processor can get the up-to-date value of the modified location.

Typically, researchers use simulation to investigate how to improve the performance of such machines. In particular, trace-driven simula-

1063-6552/97/$10.00 © 1997 IEEE

tion[3,4] offers a good trade-off between speed, accuracy, and flexibility. A key point of this methodology is to find traces (a sequence of memory references generated by the running program) that both represent typical operating conditions and include all information potentially needed for an accurate simulation of the system.[5,6] (See the sidebars for background on performance evaluation and trace generation).

We've developed a methodology and a set of tools (called *Trace Factory*) to generate traces for the performance evaluation of shared-bus, shared-memory multiprocessor systems. Trace Factory's hybrid methodology consists of

- tracing user references by means of a standard tracing tool,
- stochastically generating the kernel references, and
- simulating process scheduling and virtual-to-physical address translation.

Moreover, Trace Factory doesn't employ a standard playback of traces during the simulation phase. Instead, it uses enhanced trace-driven simulation in which the memory hierarchy conditions the reference production and the other kernel activities described above. This approach lets Trace Factory provide the accuracy crucial to the trace-generation and trace-

Performance evaluation methodologies and tools

During recent years, academic, research, and commercial groups and individuals have developed many multiprocessor performance-evaluation methodologies and tools. These tools fall into two main categories: those that help tune applications executing on a specific computer, and those that evaluate different architectural solutions while varying software features.

Important initiatives for high-performance computing, such as those from NASA, DARPA, and the NSF, have highlighted the need for tools in the first category. (See the November 1995 *Computer* and Winter 1995 *IEEE Parallel & Distributed Technology*.) Other recent tools range from simple software extensions of the processor- or operating system-monitoring software, such as PatchWrx[1], to more sophisticated and articulated environments, such as AIMS (automated instrumentation and monitoring system).[2] Companies such as DEC, Intel, CRI, and Convex have developed ATOM (analysis tools with OM), ParAide, MPP Apprentice, and CXpa, respectively, and some academic projects, such as Paradyn,[3] have become multiplatform performance-tuning software.

We focus on the second category of tools, which let computer architects tune the memory hierarchy and system parameters by selecting an arbitrary, ad hoc workload to stress the machine. The most common strategies[4] for such tools are *analytical/stochastic models*, *complete system simulation*, and *trace-driven simulation*. These strategies can be classified on the basis of different metrics, such as the accuracy of evaluation, cost of implementation, speed, and flexibility of the method over a wide range of architectures.

Analytical/stochastic models

This strategy might appear to be the most flexible and economic solution.[5] However, these models typically do not include all the aspects that characterize cache and program behavior. So, their low accuracy might be unacceptable for a deep evaluation of complex cache-based systems. This kind of analysis could be of some use for quickly estimating system performance, before completing the evaluation with a more accurate technique.

Complete system simulation

This strategy is the most flexible and accurate, because it potentially allows a detailed analysis of all the hardware and software aspects of a particular architecture, including the full operating system. Its main problem is that it requires a complete model to simulate the execution of sophisticated software such as operating systems or multiprogrammed workloads. Consequently, it usually incurs a large dilation factor, especially for high-detail simulations (such as in the SimICS[6] and SimOS[7] approaches).

In particular, for multiprocessor systems, the slowdown scales linearly with the number of CPUs being simulated. In the case of SimOS, at the deepest level of detail, a slowdown factor in the thousands occurs when simulating systems with 16 to 32 processors.[7] However, the user can control the simulation's level of detail to minimize the total simulation time.

Another tool that partially simulates the operating system activity, MINT[8] (*M*IPS *int*erpreter), provides a set of simulated processors that run standard Unix executable files compiled for a MIPS R3000-based system. MINT supports spinlocks, semaphores, barriers, shared memory, and most Unix system calls. Processors generate multiple streams of memory-reference events that drive a user-provided memory-system simulator.

Trace-driven simulation

When the performance evaluation's tar-

(*Continued on page 56*)

(*Continued from page 55*)

get is the memory hierarchy and processor-interconnection subsystem, trace-driven simulation offers a good trade-off between speed and accuracy.[9–11] This strategy produces a trace (a sequence of memory references generated by the running program) and uses it as input for a memory-hierarchy simulator. To ensure accuracy, traces must include both user and kernel references, and minimal time distortion must be induced by either the tracing mechanism (during the recording phase) or the simulator (in the utilization phase). (For more information, see the sidebar, "Trace generation.")

In a previous paper, we proposed the use of synthetic traces to evaluate and compare different system architectures by generating synthetic workloads.[9] A major advantage of synthetic over actual traces is their flexibility: a synthetic workload is much more controllable and can be used to "stress" a system. Actual-trace-driven and complete simulation can then be used to validate the results in actual conditions.

References

1. S.E. Perl and R.L. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," *Operating System Rev.*, Vol. 30, No. 11, Oct. 1996, pp. 169–183.
2. J.C. Yan and S.R. Surraki, "Analyzing Parallel Program Performance Using Normalized Performances Indices and Trace Transformation Techniques," *Parallel Computing*, Vol. 22, No. 9, Nov. 1996, pp. 1215–1237.
3. B.P. Miller et al., "The Paradyn Parallel Performance Measurement Tool," *Computer*, Vol. 28, No. 11, Nov. 1995, pp. 37–46.
4. V. Milutinovic et al., "Issues in Computer Architecture Performance Evaluation: Past Experience and Future Trends," Workshop of the Infofest '97, 1997; http://galeb.etf.bg.ac.yu/~vm/infofest97/infofest97.html.
5. M.K. Vernon, E.D. Lazowska, and J. Zahorian, "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 308–315.
6. P. Magnusson and B. Werner, "Efficient Memory Simulation in SimICS," *28th Ann. Simulation Symp.*, IEEE CS Press, 1995, pp. 62–73.
7. M. Rosenblum et al., "Complete Computer System Simulation: The SimOS Approach," *IEEE Parallel & Distributed Technology*, Vol. 3, No. 4, Winter 1995, pp. 34–43.
8. J.E. Veenstra and R.J. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," *Proc. MASCOTS '94: Second Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, IEEE CS Press, 1994, pp. 201–207.
9. 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 Systems*, Vol. 6, No. 9, Sept. 1995, pp. 915–929.
10. S.J. Eggers and R.H. Katz, "Evaluating the Performance of Four Snooping Cache Coherency Protocols," *Proc. 16th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 1989, pp. 2–15.
11. R.A. Uhlig and T.N. Mudge, "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys*, Vol 25, No. 2, June 1997, pp. 128–170.

utilization phases of the simulation[6] (see the sidebars).

Trace Factory is particularly useful for evaluating a multiprocessor architecture's performance related to different workloads and most of the influencing activities of the operating system. The designer can evaluate and tune architectural solutions for coherence protocol, cache structure, bus, and memory.

The Trace Factory environment

Trace Factory is an operating environment to create traces representing a specific user workload executing on a specific multiprocessor configuration with a particular kernel behavior. Starting with a set of *source* traces including only user references, Trace Factory can produce complete multiprocessor *target* traces. Source traces can be obtained through a tool based on the same microprocessor that the target system uses.

Trace Factory generates target traces by considering the source traces, the target-machine configuration (for example, the number of processors), and the three kernel aspects that most affect global performance:

- *Kernel memory references*—the reference bursts caused by each system call and kernel-management routine.
- *Process scheduling*—the dynamic assignment of a ready process to an available processor.
- *Virtual-to-physical address translation*—the mapping of virtual addresses produced by a running process to physical memory addresses.

Trace Factory can simply store the reference sequences into target-trace files or can supply them to the multiprocessor simulator via synchronous channels. In the latter case, Trace Factory generates the target trace according to the *on-demand* policy: It produces a new reference when the simulator requests one, so that the temporal behavior imposed by the memory subsystem conditions the generation of the trace.

Generation of kernel references

Kernel references affect performance because they interrupt the locality of the memory references of the running process, causing additional cache misses. Our hybrid approach models the kernel-reference stream through a stochastic model of references and bursts.[4]

Trace Factory obtains a process-reference stream by inserting sequences of kernel references (kernel bursts) in the user-reference stream (source trace). For each kernel reference, Trace Factory produces the area referenced (code or data), the address within the

Trace generation

Tracing techniques include hardware and software solutions. Further classification is problematic, because each tool has at least one feature that makes it unique and precludes it from being grouped with others.

Hardware solutions

Hardware monitoring potentially provides the highest accuracy. With this technique, Bart Vashaw and Drew Wilson have used two identical machines (Encore Multimax 320s) to collect traces.[1] Their procedure records at full speed the references generated by the traced machine into the tracing machine's memory until the trace memory is exhausted. Then, the tracing machine starts storing the traces. The procedure inserts time stamps at synchronization points to allow the correct replay of the collected traces. Its main advantages are accuracy and the absence of time distortion and intrusiveness. Its most critical drawback is that modern processor technology encourages the adoption of on-chip caches, so that a large number of memory references are handled internally and can no longer be captured by the hardware tracing mechanism. Other limiting factors are the high cost of implementation and the lack of completeness (fragmentation) of the trace gathered, because of the limited size of storage buffers. Finally, traces obtained from a multiprocessor machine through hardware techniques cannot be employed for an exhaustive performance analysis of the system, because producing traces with a variable number of processors is impractical.

Software solutions

Software tracing methods include *program instrumentation*, *single-step execution*, and *microcode modification*. These methods all suffer to varying degrees from *time dilation* because their tracing mechanisms generate a heavy overhead. This overhead drastically changes the relative timing of asynchronous events, resulting in lower accuracy compared to hardware approaches.

Program instrumentation

This method adds a set of instructions to create at runtime the portion of the trace relative to each *basic block* (a sequence of machine-level instructions not containing branches) throughout the program. The instrumentation phase can be activated at either the source or executable level. The latter is simpler for the user to handle, but it is difficult to implement and might not allow instrumentation of all programs. On the other hand, instrumentation at the assembly level is easy, but the user must have access to the entire source code. Also, the lack of completeness in the trace limits the model's accuracy, because capturing references of kernel routines is difficult.

Mptrace,[3] Trapeds,[4] and TangoLite[5] are tracing tools based on program instrumentation. Mptrace, developed by Susan J. Eggers and her colleagues for the Sequent i-386 shared-memory multiprocessor, collects traces of multithreaded parallel programs. It automatically modifies the assembly-

(*Continued on page 58*)

selected area, and the kind of access (read or write).

To obtain the parameters that describe our model, we use a trace that includes kernel references. In particular, we directly evaluate the probability of code or data access and of data read or write access by counting the relative occurrences of events. Concerning the locality of memory references, we evaluate three parameters for both code and data areas:

- the maximum distance between two consecutive references,
- the maximum amplitude of the distribution of distances between two consecutive references, and
- the percentage of backward references over the total number of nonsequential accesses.

We use these parameters to set up the shape of an empirical function that gives, step by step, the next address to be inserted into the synthetic kernel-reference stream.[4]

Two statistics govern the generation of kernel bursts: the length of each burst and the distance between the starting point of two consecutive bursts. We measure the distribution of the kernel-burst length and the distance between the beginning of two successive bursts in real traces.

If the tracing tool records the system-call positions, the burst insertion will be driven by this information collected in the source traces. This lets us generate more accurate workloads (for example, considering that the processes typically exhibit a different number of system calls).

This hybrid methodology introduces approximations concerning both the generated address and the distribution of kernel bursts. The weight of such error appears to be somewhat limited, considering that our goal is the trace generation for performance evaluation of the memory subsystem.

To estimate the error induced by the synthetic generation of the kernel-reference stream, we consider a series of eight processor traces distributed by Carnegie Mellon University and obtained on an Encore Multimax (shared-bus multiprocessor) machine (see Table 1). These traces represent a wide variety of application domains; they include both user and kernel references. The Multimax ran a version of Carnegie Mellon's Mach operating system.

Table 2 includes the kernel-access percentages (code, data, and write) and the statistics concerning the distribution of burst distance and lengths. The table summarizes the distance and length by average value (μ) and standard deviation (σ).

We employed trace-driven simulation to compare the results of four situations:

(*Continued from page 57*)

language version of the application, inserting code to collect traces. Craig Stunkel and W. Kent Fuchs originally developed Trapeds for the Intel iPSC/2 hypercube multicomputer. Their original version traces both user and kernel code and performs execution-driven simulation to avoid large storage costs. They implemented a later extension on the Encore Multimax 510, an eight-node bus-based multiprocessor. To guarantee the accurate recreation of the interactions between processors, Trapeds uses a timer-based approach. TangoLite, developed by Stephen E. Goldschmidt and his colleagues, is a software instrumentation system for the MIPS instruction architecture. It supports the execution-driven simulation of multiprocessor workloads, and it can generate multiprocessor traces. TangoLite performs the instrumentation mostly at the assembly level. It represents the processes as lightweight threads, and its scheduling policy guarantees the chronological simulation of events generated by different processors.

Single-step execution

This method works with microprocessors that allow a program's execution to be interrupted after each instruction. Because kernel routines typically disable interruptions, this technique usually cannot capture references generated by the execution of kernel routines. In the trace generator implemented by Eggers and Randy Katz for the Sequent architecture,[6] the tracing mechanism uses trace-trap facilities to halt at each instruction and dump trace information, both for instructions and their operands.

Microcode modification

This technique can be considered as either a hardware or a software methodology. We have chosen to classify it as a software solution because it causes a slowdown for the processor, like other traditional software techniques. ATUM (address tracing using microcode) uses processor microcode to record references in a reserved part of the main memory as a side effect of normal execution.[7] Compared with other techniques, it leads to fewer distortions and very fast recording (only 10× slowdown). All the system activities can be observed, with no additional required hardware. Its disadvantages include poor flexibility, and the trace length is limited to the amount of the memory reserved for the trace storage.

WHICH SOLUTION TO USE?

When the goal is to compare different architecture solutions, analyzing the system behavior under predefined and controlled workloads becomes important. So, for trace generation to be effective,

- traces must represent actual workloads for the target machine, and
- the designer must be able to produce proper traces to investigate the system's behavior in specific (possibly critical) workload conditions.

Because only software techniques can guarantee this kind of flexibility, we have conducted our research in that direction.

References

1. B. Vashaw, "Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors," Tech. Report CMUCDS-93-4, Dept. of Electrical and Computer Eng., Carnegie Mellon Univ., Pittsburgh, 1993.
2. S.J. Eggers et al., "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," *Proc. SIGMETRICS '90: Ann. Conf. Measurement and Modeling of Computer Systems*, ACM Press, New York, 1990, pp. 37–47.
3. C.B. Stunkel, B. Janssens, and W.K. Fuchs, "Address Tracing of Parallel Systems via Trapeds," *Microprocessors and Microsystems*, Vol. 16, No. 5, June 1992, pp. 249–261.
4. S.R. Goldschmidt, *Software Coherence in Multiprocessor Memory Systems*, PhD thesis, Stanford Univ., Computer Systems Laboratory, Stanford, Calif., 1993.
5. S.J. Eggers and R.H. Katz, "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 373–382.
6. R.L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 1988, pp. 186–195.

Table 1. The CMU multiprocessor traces.

APPLICATION	SOURCE	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
Pde	A. Wilson	Partial differential equation solver

a. The original CMU traces.
b. The CMU traces deprived of the kernel references.
c. The original traces, with kernel references replaced by a reference stream generated synthetically on the basis of each original trace's statistics, yet preserving the same position and length of each burst as in the original traces.
d. The original traces, with kernel references generated synthetically on the basis of the MP3D statistics, yet preserving the same position of each burst as in each original trace.

In all these situations, we measure global system power as $GSP = \sum U_{cpu}$, where

$$U_{cpu} = \frac{T_{cpu} - T_{delay}}{T_{cpu}} \times 100$$

and T_{delay} is the total CPU delay time caused by waiting for memory-operation completions. The GSP depends

Table 2. Kernel-reference statistics.

Applications	Kernel references (%)	Kernel burst: Distance μ	Distance σ	Length μ	Length σ	Kernel code references (%)	Kernel data: References (%)	Writes (%)
Ecas	3.32	27,586	793	928	1,288	2.12	1.21	0.45
Hartstone	8.47	4,004	9,261	341	1,421	5.36	3.11	1.05
Locusroute	6.93	20,214	12,037	1,404	2,430	3.96	2.97	1.29
MP3D	3.21	28,357	901	911	1,134	2.05	1.17	0.43
Pde	5.30	21,805	11,369	1,158	2,189	3.40	1.90	0.75

on both bus utilization and the traffic portion caused by cache misses and coherence operations. For this reason, Tables 3 and 4 show all these metrics—that is, GSP, bus utilization, miss rate, and write transactions.

Tables 3 and 4 show error percentages with respect to the values obtained from the actual traces. Columns *a* to *d* correspond to the four situations listed above. The simulations use 64-byte blocks and the Dragon coherence protocol.[2] The timing parameters for the 64-bit bus are the same as in the example we'll discuss in the next section (see Table 7, Timing I). We analyzed a sample trace of 2,500,000 references per processor.

Table 3 shows the results of simulations for a 256-Kbyte, direct-mapped cache. Although both the miss rate and the number of bus transactions incur large errors, the GSP values show a relatively low error, because the system has low bus utilization (≃ 45%). The percentage of error for the GSP increases with high bus utilization. Table 4 shows the results of a simulation for a smaller cache size (64 Kbytes), with 65.5% average bus utilization.

Table 3. Kernel model validation—low bus utilization (≅ 45%).

	Application	Actual *a*	Error (%) *b*	*c*	*d*
Global system power	Ecas	684.8	+3.6	–1.8	–1.1
	Hartstone	780.4	+2.1	+0.1	–0.7
	Locusroute	723.8	+7.2	–0.8	+4.0
	MP3D	632.1	+4.8	–0.1	–0.1
	Pde	780.5	+1.5	0.0	–0.2
	Average square error		4.4	0.9	1.9
Bus utilization (%)	Ecas	65.3	–3.7	+3.4	+2.9
	Hartstone	18.4	–72.8	–11.1	+6.0
	Locusroute	44.6	–53.8	–2.7	–27.5
	MP3D	71.1	–6.7	+1.1	+1.1
	Pde	23.1	–35.0	–1.7	–2.1
	Average square error		43.5	5.5	12.7
Miss rate	Ecas	0.273	–17.5	+3.6	+1.5
	Hartstone	0.071	–77.0	+14.0	+42.2
	Locusroute	0.292	–61.3	+1.0	–29.8
	MP3D	0.522	–9.4	+1.1	+1.1
	Pde	0.100	–40.0	+20.0	+24.0
	Average square error		48.2	11.1	25.5
Write transactions per 1,000 memory operations	Ecas	47.50	+6.3	+4.6	+5.0
	Hartstone	10.58	–67.2	–19.2	+3.4
	Locusroute	8.44	–19.9	–13.6	+11.9
	MP3D	0.88	–85.2	+23.8	+23.8
	Pde	13.20	–27.2	–7.5	–21.0
	Average square error		51.0	15.5	15.4

In any case, the insertion of a stochastically generated kernel-reference stream reduces the error, compared to traces not including kernel references (compare columns *c* and *d* to column *b*). The insertion of kernel references derived from statistics collected from other traces (MP3D) introduces a lower error in respect to traces consisting of only user references (compare columns *d* and *b*). This shows that our methodology is useful when we have only a tracing tool that cannot capture kernel references. Our methodology's accuracy concerning the memory subsystem also depends on the behavioral differences between the workload from which the kernel statistics are extracted and the applications to which the synthetic kernel-generation model is applied.

Process management

One main goal of the multiprocessor scheduler is to provide an acceptable degree of load balance to let the programmer develop applications without caring about the load distribution on the processors. Nevertheless, load balance induces process migration that causes further coherence overhead. That is, the migration of a process can replicate in more than one cache a memory block belonging to a private area of that process. The coherence protocol treats these copies

Table 4. Kernel model validation—high bus utilization (65.5%).

	Application	Actual	Error (%)		
		a	b	c	d
Global system power	Ecas	357.1	+9.6	+3.1	+3.4
	Hartstone	765.7	+3.5	+0.2	–1.2
	Locusroute	555.6	+30.2	+6.3	+20.2
	MP3D	420.4	+15.0	+3.7	+3.7
	Pde	740.1	+4.7	+0.3	+1.3
	Average square error		15.9	3.5	9.3
Bus utilization (%)	Ecas	95.1	–2.0	–0.7	–0.6
	Hartstone	23.9	–71.1	–13.1	+9.0
	Locusroute	81.6	–51.5	–11.6	–31.3
	MP3D	94.7	–4.1	–1.0	–1.0
	Pde	37.2	–39.1	–3.5	–10.2
	Average square error		43.0	8.0	15.3
Miss rate	Ecas	1.247	–6.3	–3.4	–3.8
	Hartstone	0.130	–77.7	–4.6	+28.4
	Locusroute	0.768	–63.3	–22.6	–45.8
	MP3D	0.891	–15.0	–4.9	–4.9
	Pde	0.225	–45.8	–0.8	–6.2
	Average square error		49.8	10.6	24.4
Write transactions per 1,000 memory operations	Ecas	35.63	+1.4	–0.1	0.0
	Hartstone	8.43	–58.8	–28.8	–19.0
	Locusroute	4.63	–36.5	–32.2	+27.9
	MP3D	0.67	–81.6	–37.6	–37.6
	Pde	9.75	–16.7	–7.5	–27.0
	Average square error		48.4	25.8	25.6

as shared. Such *passive sharing*[7] or *process-migration sharing*[1] results in a heavy and useless burden for the shared bus. Furthermore, on every context switch, a burst of cache misses occurs, because of the loading of the working set of the new process. A scheduling policy based on *cache affinity* can reduce the effects of process migration and context switches.[8] Cache-affinity scheduling tries to schedule a process on the processor where it last executed, to minimize cache misses after a context switch by reusing the blocks recently stored in the local cache.

Trace Factory models process management by simulating a simple scheduler. The input parameters for the scheduler are the number of processes (N_{proc}), the number of processors on the target machine (N_{cpu}), the time slice in terms of number of references (T_{slice}), the process-scheduling policy (cache affinity or *random*), and the process-activation policy (*nonblocking* or *two-phase*). To simulate the scheduler, Trace Factory

- starts from a set of source traces including a synthetic kernel (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.

By using the on-demand policy, the speed of each simulated processor and the memory hierarchy influence the scheduling activity, just as in real systems.

For uniprocess applications, the scheduler operates as follows. If a process p is running on processor P for a D time interval (specified in terms of number of references), D references of the p source trace become references for processor P. At the simulation start-up, all the processes are ready and are inserted in a proper queue, R_1. Initially, the scheduler randomly selects N_{cpu} processes, and each running process has a different time slice (the process running on processor i is assigned a time slice $T_i = I \cdot T_{slice}/N_{cpu}$). On each processor, after the first context switch, the next scheduled process is regularly assigned T_{slice}. This strategy, typically adopted in multiprocessor operating systems, avoids a context switch being needed simultaneously on all processors every T_{slice}. Such a situation would produce an overlap of miss peaks on all caches. This overlap would cause bus saturation because of the bus transactions needed to fetch missing blocks from memory.

On a context switch, the scheduler extracts a process from R_1 and assigns the process to the available processor. Process selection can follow the cache-affinity policy or can just be random. The scheduler can manage the preempted process in two ways. The nonblocking activation policy immediately inserts the preempted process into R_1. This strategy potentially suffers from *starvation*: a target trace might not include the references of a process, when the trace's length is short and $N_{proc}/N_{cpu} \gg 1$. The two-phase activation policy uses another queue, R_2, which is initially empty (see Figure 1). On every context switch, the scheduler inserts the preempted process into R_2 (phase one). As soon as R_1 becomes empty, the scheduler takes all the processes from R_2 and inserts them into R_1 (phase two). This strategy ensures that a process does not have to wait an indefinite time for its turn. Indeed, a process cannot execute $n + 1$ times before each other process executes n times.

For multiprocess applications, source traces must include synchronization tags representing the actual

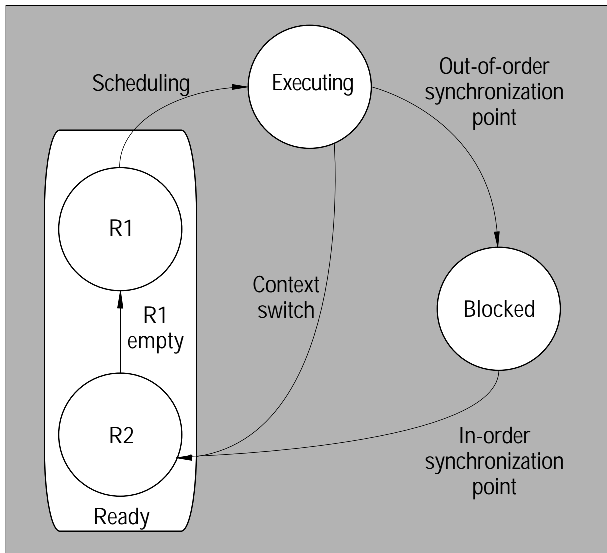

Figure 1. State-transition diagram for the two-phase activation strategy.

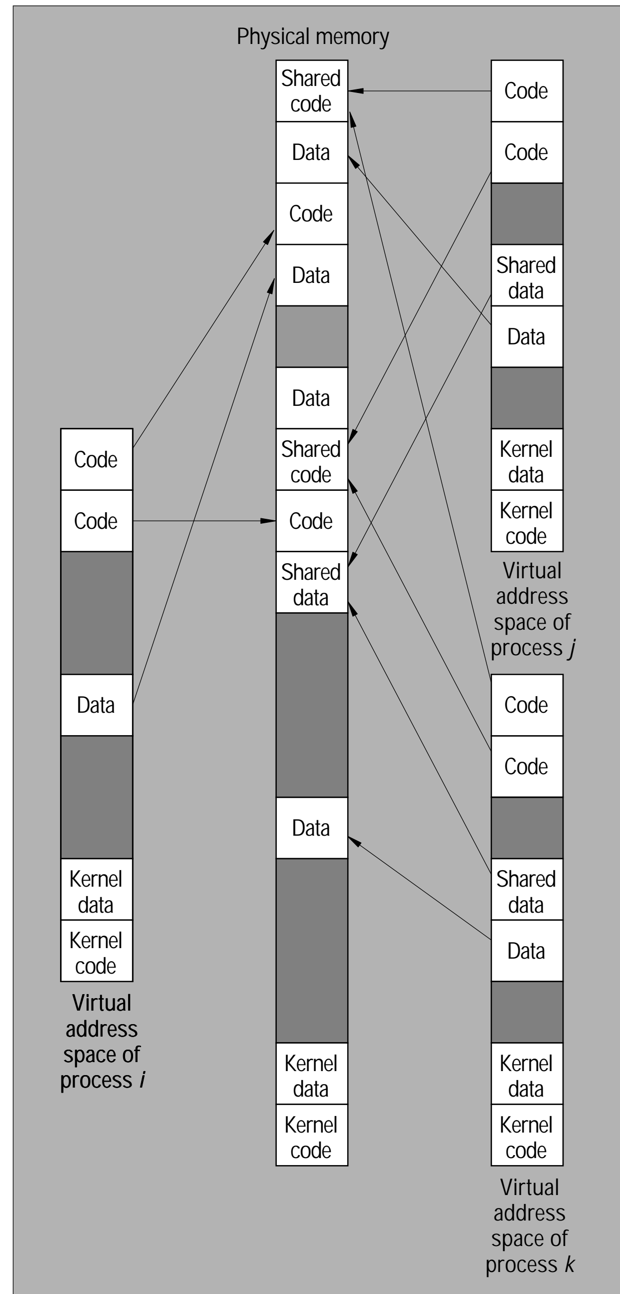

Figure 2. A scheme of the virtual-to-physical address translation.

synchronization sequence of the parallel-application execution.[9] In this case, the process scheduling is driven by the time slice and the synchronization sequence for multiprocess applications. When a process reaches an out-of-order synchronization event, it is inserted into the Blocked waiting queue (see Figure 1) to wait for the synchronization event. Then, it enters either R_1 or R_2, depending on the activation policy.

Virtual-to-physical address translation

In virtual-memory models based on paging, a running process might produce virtual and physical references that have different localities. The mapping of sequential virtual pages into nonsequential physical pages causes this difference and influences the number of *intrinsic-interference* (or *conflict*) *misses* caused by interferences among kernel code and data, user data, and code accesses in the same cache set.

We model the virtual-to-physical address translation as follows (see Figure 2). We suppose that each process has its own address space for code and private data. The kernel and its associated data structures reside at the bottom of the virtual address space of each process. When a set of processes shares a memory area, the system ensures that for such processes, the shared area is mapped on the same set of physical-memory pages. When a number of instances of the same application are active, their code is shared. Finally, kernel instances share a unique set of physical memory pages.

Each process starts its execution without any page being stored into physical memory in advance. As soon as a process tries to operate a location in a page that does not reside in the physical memory, a page fault is generated and the pertinent page is allocated (on-demand paging). This page fault triggers a context switch, and the process is suspended from execution and spends a predefined number of cycles in the Blocked waiting queue, as is the case for any other synchronization event. In this way, we model the delay needed to fetch the required page from disk. Because the traces usually lack any information concerning I/O operations, no other aspect of I/O interaction is considered.

Using Trace Factory

We used Trace Factory to evaluate and compare five coherence protocols for a shared-bus, shared-memory multiprocessor, using three typical workloads and two bus timings. We evaluated the protocols' performance

Table 5. Statistics of uniprocess application and Unix command traces.

Application	Distinct blocks	Code (%)	Data (%) Read	Data (%) Write	System calls
awk (beg)	4,963	76.76	14.76	8.47	29
awk (mid)	3,832	76.59	14.48	8.93	47
Cjpeg	1,803	81.35	13.01	5.64	18
cp (beg)	2,615	77.53	13.87	8.60	26,526
cp (mid)	2,039	78.60	14.17	7.23	56,388
Msim	960	84.51	10.48	5.01	345
dd	139	77.47	16.28	6.25	47,821
Djpeg (beg)	2,013	81.00	12.75	6.26	15
du	1,190	75.86	16.37	7.77	9,474
lex	2,126	78.67	15.49	5.84	40
Gzip	3,518	82.84	14.88	2.28	13
ls -aR	2,911	80.62	13.84	5.54	1,196
ls -itR (beg)	2,798	78.77	14.58	6.64	1,321
ls -itR (mid)	2,436	78.42	14.07	7.51	1,778
rm (beg)	1,314	86.39	11.51	2.10	10,259
rm (mid)	1,013	86.29	11.65	2.06	15,716
Telnet (beg)	781	82.52	13.17	4.31	2,401
Telnet (beg)	205	82.78	12.93	4.28	2,827

for global system power and bus utilization, because for this kind of machine, the shared bus is the bottleneck that limits the architecture's scalability.

Source-trace production

Typical workloads for a multiprocessor workstation consist of a set of Unix commands, uniprocess applications, and multiprocess applications. We selected some typical Unix commands (**awk**, **cp**, **dd**, **du**, **lex**, **rm**, and **ls**) with different command-line options, three utility programs (**Cjpeg**, **Djpeg**, and **Gzip**), a network application (**Telnet**), and a user application (**Msim**, the multiprocessor simulator used in this work). In a typical situation, various users might run different system commands and ordinary applications. To take into account that users can use the same program at different times, we traced some commands in shifted execution sections: initial (*beg*) and middle (*mid*). Table 5 describes these source traces in terms of the number of distinct (unique) blocks the program uses; code, data-read, and data-write access percentages; and the number of system calls.

Because the sharing pattern of parallel applications significantly influences a multiprocessor's performance, we considered two parallel programs with different sharing behavior, **MP3D** and **Cholesky**, both from the Splash suite. **MP3D** simulates rarefied hypersonic flow; the generated trace relates to a case of 10,000 molecules and 20 time steps. **Cholesky** factorizes a sparse positive definite matrix, using the homonymous method. For **Cholesky**, we generated the trace using as input a 1,806 × 1,806 matrix with 30,284 nonzero elements coming from the Boing/Harwell sparse matrix test (bcsttk14). We used TangoLite (see the sidebar, "Trace generation") to produce all source traces. Thus, the traces belong to MIPS-based machines. We traced the parallel applications on a virtual multiprocessor having as many processors as the number of application processes.

Table 6. Statistics of multiprocess source traces. WRR is write-run length and XRR is external rereads.

Workload	Processors	Distinct blocks	Code (%)	Data (%) Read	Data (%) Write	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	5,173	78.14	15.22	6.64	913	9.10	2.22	11.88	10.83	2.15	2.65
	4	6,480	78.56	14.30	7.14	1,625	10.34	3.20	8.18	6.04	1.54	1.60
	6	6,923	78.70	13.99	7.31	2,004	10.91	3.56	7.03	5.06	1.51	1.63
	8	7,169	78.77	13.84	7.39	2,309	11.30	3.76	6.55	4.65	1.50	1.67
	10	7,308	78.81	13.74	7.45	2,597	11.62	3.91	6.25	4.40	1.50	1.71
	12	7,397	78.83	13.68	7.49	2,820	11.88	4.02	6.07	4.32	1.50	1.73
	14	7,509	78.85	13.65	7.50	3,002	12.07	4.10	5.89	4.18	1.51	1.75
Cholesky	2	14,312	79.35	12.88	7.77	1	0.14	0.00	2.00	0.00	2.00	0.00
	4	17,119	79.83	13.57	6.60	7,215	8.29	1.19	4.75	3.47	1.06	0.65
	6	19,172	80.21	13.65	6.14	8,789	9.67	1.32	4.46	3.12	1.06	0.68
	8	20,569	80.43	13.66	5.91	10,079	10.24	1.36	4.43	3.04	1.05	0.63
	10	21,557	80.69	13.70	5.61	11,268	10.82	1.44	4.54	3.00	1.06	0.74
	12	22,900	80.98	13.69	5.33	12,404	11.18	1.44	4.89	3.67	1.05	0.61
	14	23,669	81.11	13.70	5.19	12,876	11.47	1.48	5.10	3.80	1.05	0.63

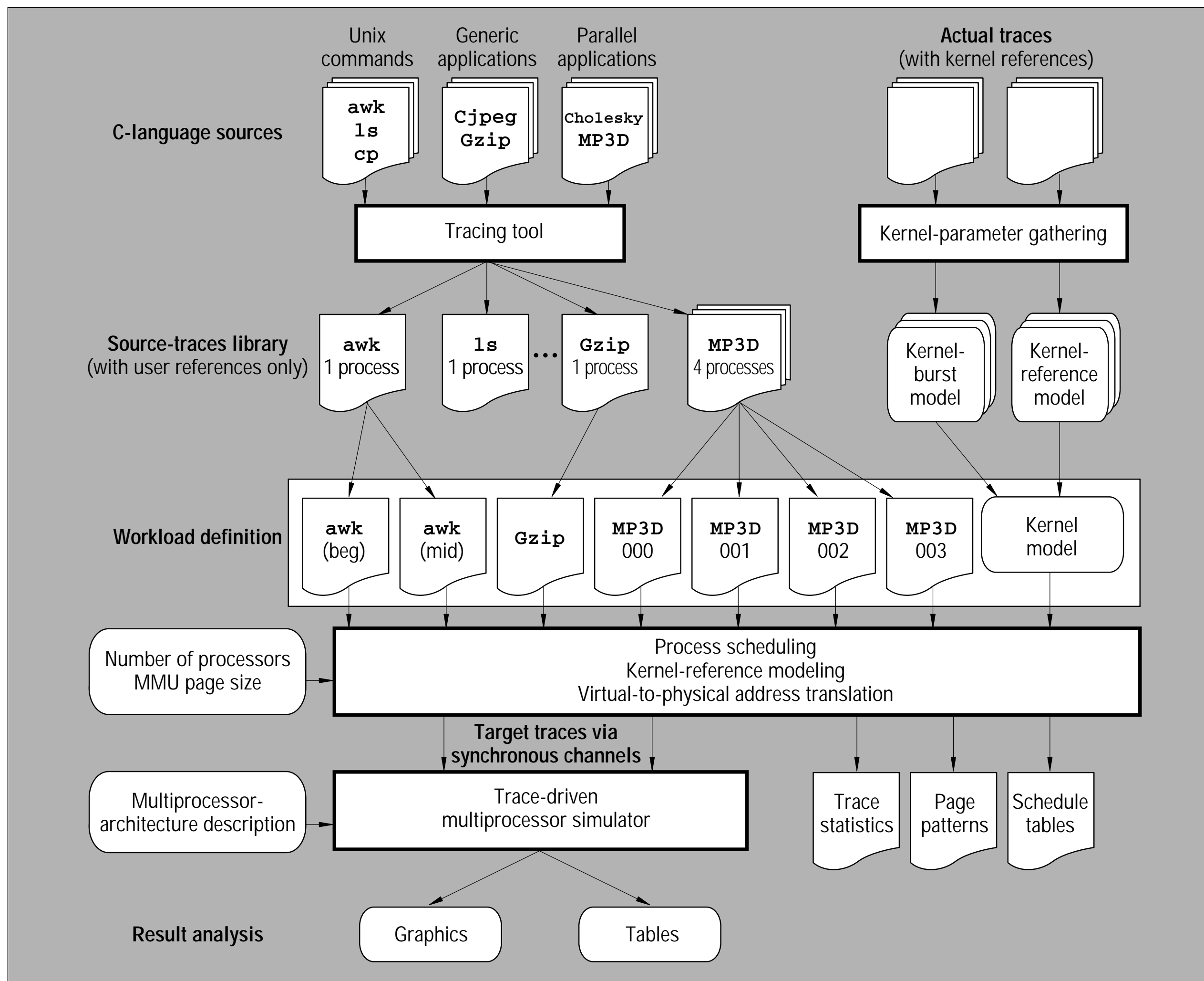

Figure 3. The global scheme to produce and use a target trace in a simulation.

Table 6 summarizes the statistics for the multiprocess application traces. Two metrics—*write-run length* (WRL) and *external rereads* (XRR)—characterize access patterns to shared data.[3] The first is the number of write operations from a given processor to a memory block before another processor accesses that block. (A *write-run* is the sequence of write—eventually interleaved by read—references.) The second metric indicates how many read operations will use a block after one write-run has terminated and before another starts. A natural use of write-run statistics is to select the better coherence strategy between *write-invalidate* and *write-update* for a given workload. A long write-run suggests that a write-invalidate coherence protocol should be chosen. The cost of the initial miss (caused by invalidation) is balanced by the large amount of bus traffic saved because all the subsequent write operations can execute locally during the write-run. A large number of external rereads indicates to what extent different processors need a copy and, therefore, whether a write-update strategy would be convenient.

The write-run length also indicates whether a specific shared address exhibits fine-grain sharing, or whether each processor uses that address sequentially over long periods of time.[3] For our test case, the statistics show that

- `MP3D` exhibits coarse-grained sharing, because the average write-run length varies from 5.89 to 11.88; and
- `Cholesky` exhibits medium-grained sharing, having an average write-run length from 2.00 to 5.10.

Target workload production and simulated-architecture parameters

The source traces considered in the previous paragraph are unscheduled; they do not include kernel references; and the references involve virtual addresses. For our protocol analysis, we generated three workloads: *UniP*, *Mix1*, and *Mix2*. UniP incorporates 30 uniprocess applications from Table 5. Mix1 and Mix2 consist of 30 uniprocess applications and the load from a parallel application (`MP3D` and `Cholesky`, respectively) that generates a number of processes equal to one-half of the processors available on the machine (see Table 6).

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

Class	Parameter	Timing I	Timing II
CPU	Read cycle	2	2
	Write cycle	2	2
	Duration of each time interval (cycles)	14	14
	Maximum number of references per time interval	2	2
	Probability of 0 references per time interval	0.1	0.1
	Probability of 1 reference per time interval	0.3	0.3
	Probability of 2 references per time interval	0.6	0.6
Cache	Cache size	256 Kbytes	256 Kbytes
	Block size	64 Kbytes	128 Kbytes
	Associativity	2	2
	State updating	1	1
	Write cycle	1	1
	Read cycle	1	1
Bus	Width	64 bits	64 bits
	Write transaction	5	5
	Invalidation	5	5
	Update-block transaction (UpdateB)	18	34
	Memory-to-cache read-block transaction (MReadB)	32	48
	Cache-to-cache read-block transaction (CReadB)	22	38

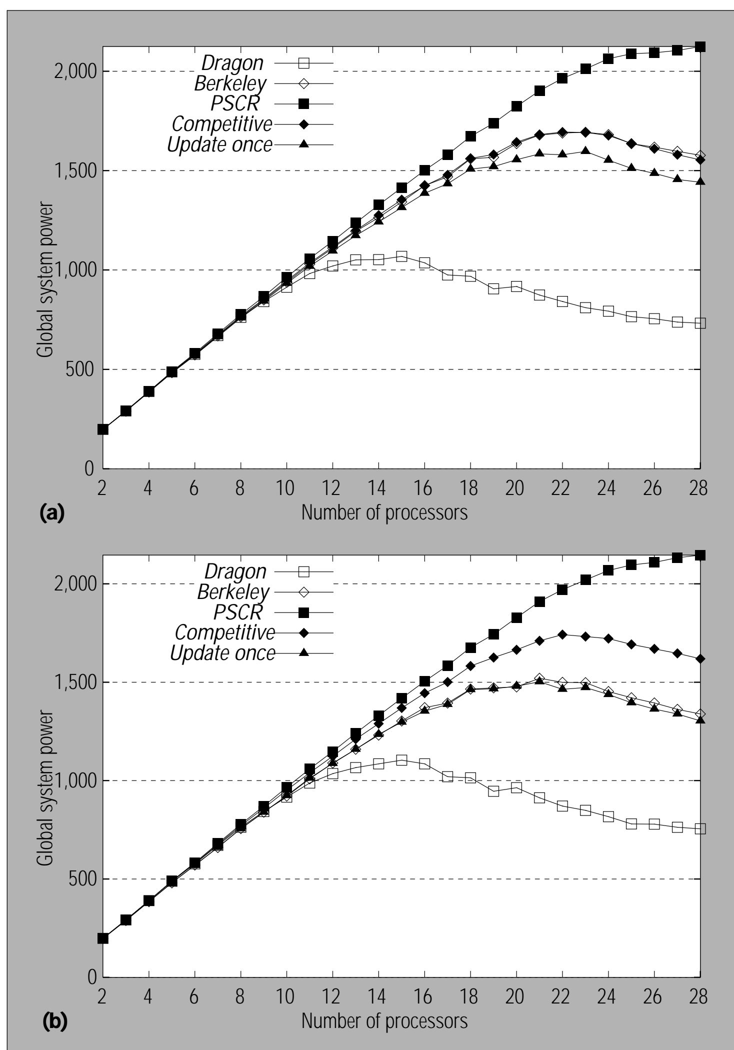

Figure 4. Global system power for the UniP workload: (a) Timing I; (b) Timing II.

Figure 3 shows the scheme for producing a target trace for Mix1.

The simulator[4] used for our analysis characterizes a multiprocessor in terms of CPU, cache, and bus parameters (see Table 7). The CPU parameters are the clock cycle, the minimal number of clock cycles for a read/write operation, and the temporal distribution of the memory accesses. We describe this distribution in terms of the maximum number (M) of references per time interval and the probability that this interval contains exactly 0, 1, 2, …, M memory references. That time interval is a fixed number of CPU clock cycles.

The cache parameters are cache size, block size, associativity, and the duration of a read/write operation on a cached copy. The simulator employs a least recently used (LRU) replacement policy. We can simulate eight different bus-based coherence protocols.

Finally, the bus parameters are the number of CPU clock cycles for each kind of transaction: write, invalidation, update-block, and memory-to-cache and cache-to-cache read-block.

The target architecture of our analysis is a multiprocessor consisting of a set of MIPS-R3000-like independent processors (ranging from two to 28). Each processor has a 256-Kbyte, two-way set-associative cache. The processors access shared memory via a 64-bit shared bus. The page size is 4 Kbytes; the time slice is 200,000 references; and the analyzed execution time corresponds to 2,500,000 references per processor. For choosing a process on a context switch, we adopted random selection from the ready queue, with two-phase activation. The simulations involved two block sizes: 64 bytes and 128 bytes, which generated two different sets of timings for bus transactions (see Table 7).

EVALUATING COHERENCE PROTOCOLS UNDER DIFFERENT WORKLOADS

We used the target traces we've described to analyze the behavior of coherence protocols as a function of the workload features. We chose these protocols: *Dragon*, *PSCR*[10] (passive-shared-copy removal), *Berkeley*, *Update*

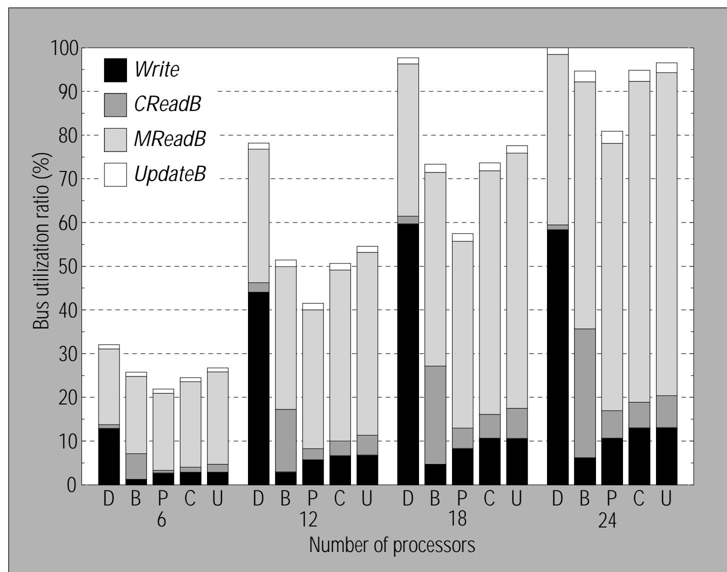

Figure 5. The bus-utilization ratio, segmented by transaction type, for the UniP workload. For each processor configuration, the figure shows the behavior of the five protocols (D = Dragon, B = Berkeley, P = PSCR, C = Competitive Snoopy Caching, and U = Update Once). The Write bar also includes invalidation transactions for those protocols that use this kind of transaction.

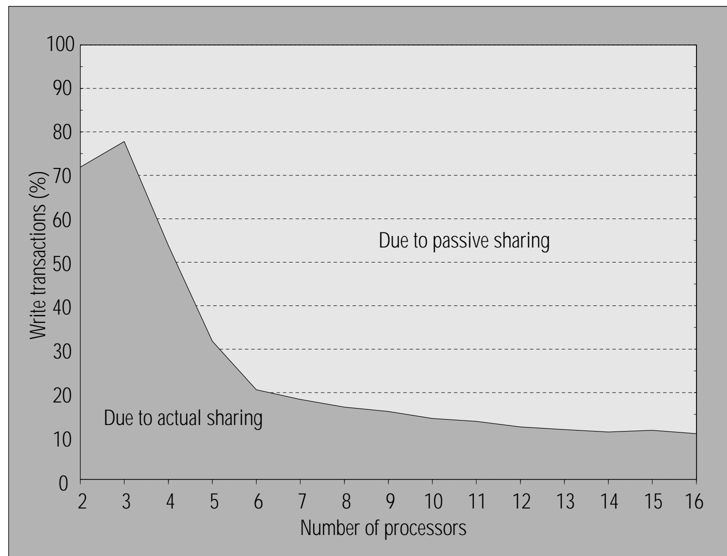

Figure 6. Write transactions involving passive shared copies for the UniP workload, Dragon protocol, and Timing I.

Once,[11] and *Competitive Snoopy Caching*. (A book by Milo Tomasevic and Veljko Milutinovic provides background on Dragon, Berkeley, Competitive Snoopy Caching, and other protocols and aspects related to cache coherence.[2]) Dragon uses a write-update (write-broadcast) strategy, whereas the other protocols use an invalidation strategy. PSCR employs *selective invalidation* to limit the number of passive shared copies. It invalidates the copies belonging to the private data area of a process as soon as another processor fetches them. Berkeley and Update Once invalidate on the first or 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.

Let's first examine the UniP workload, in which actually shared areas belong only to the kernel, and further shared copies are generated because of process migration. These passive-shared copies are the main percentage of total shared copies. As we expected, PSCR performs best (see Figure 4), because it systematically destroys these passive shared copies. For the same reason, all the protocols that invalidate shared copies perform better than Dragon. We used the two block sizes to show how the different bus timings influence performance.

Dragon performs poorly compared to the other protocols because it causes bus saturation for a lower number of processors (see Figure 5). Much of this load consists of write transactions on shared copies, which are caused by process migration. Figure 6, which shows the percentage of write transactions on passive shared copies for Dragon, highlights the passive-sharing phenomenon.[7]

Berkeley, Update Once, and Competitive Snoopy Caching achieve fewer write transactions and invalidations by using other invalidation policies. These indiscriminating invalidation strategies, which act on all shared copies, might cause a *miss raise* (an increase in misses) on account of active shared-copy invalidations. Those protocols use a portion of the bus bandwidth for read-block transactions and a much smaller portion for write transactions and invalidations. In particular, Berkeley's invalidation reduces the amount of shared copies and thus the number of invalidation transactions. At the same time, this strategy causes a miss increase whose consequences are less important because the protocol heavily employs the cheaper cache-to-cache transactions.

Figure 7 shows the system behavior for Mix1 and Mix2. The introduction of a parallel application with coarse-grain sharing (Mix1) decreases the global

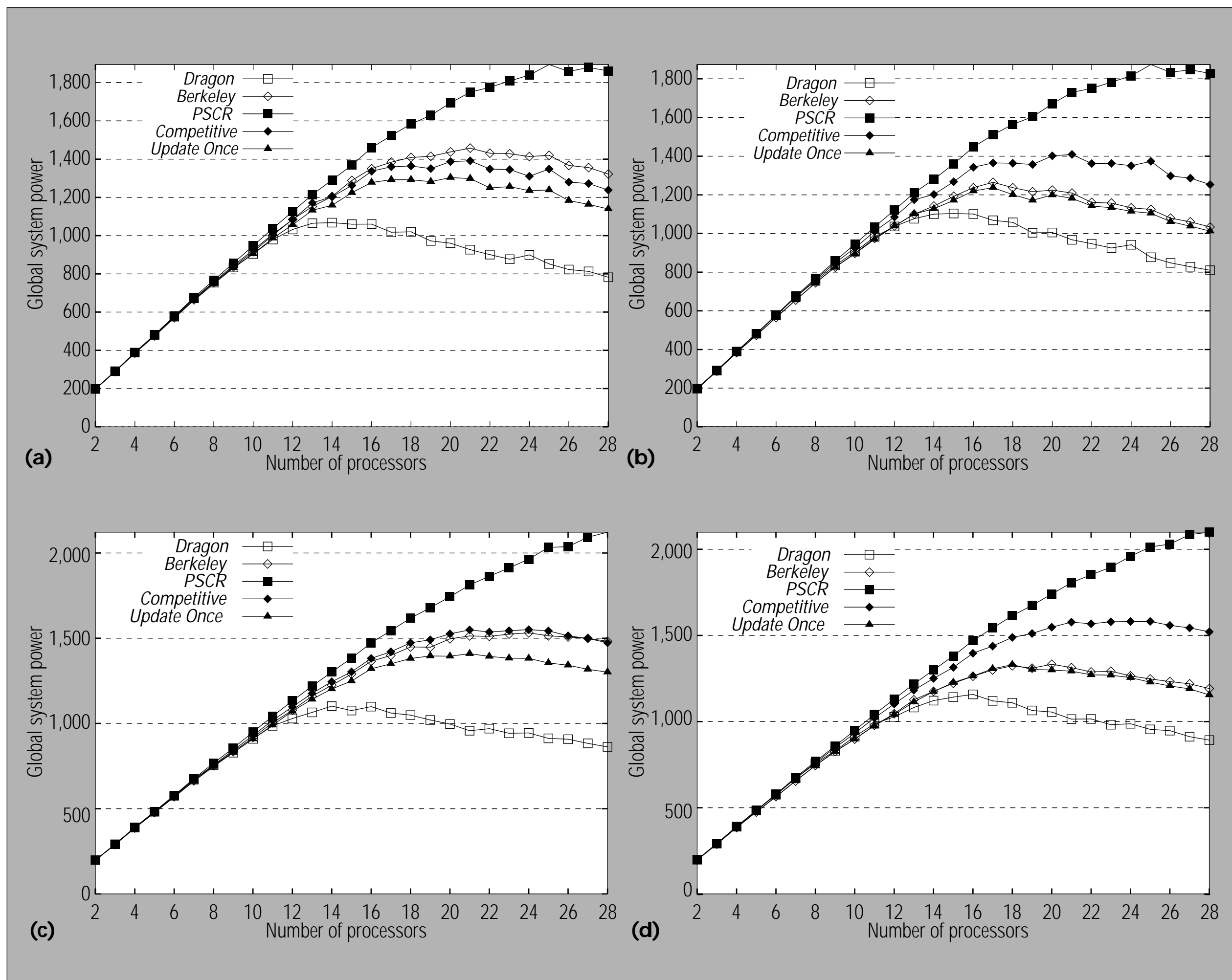

Figure 7. Global system power: (a) Mix1, Timing I; (b) Mix1, Timing II; (c) Mix2, Timing I; (d) Mix2, Timing II.

Table 8. Statistics of target traces for Dragon with Timing 1. WRR is write-run length and XRR is external rereads.

Workload	PEs	Distinct blocks	Code (%)	Data (%)		Shared blocks	Shared data		Write-run			
									WRL		XRR	
				Read	Write		Accesses	Write	μ	σ	μ	σ
UniP	8	65,989	77.00	13.52	9.48	9,912	16.52	4.96	20.63	10.48	5.30	8.84
	12	97,218	77.10	13.26	9.64	13,577	17.24	5.32	20.05	9.83	6.32	9.30
	16	125,186	77.10	13.24	9.66	15,692	17.57	5.44	19.69	10.64	5.82	9.04
	20	183,987	77.28	13.07	9.65	17,249	17.63	5.45	19.14	11.21	5.76	9.01
	24	257,916	77.52	12.80	9.68	19,954	17.32	5.40	18.54	11.19	5.09	8.40
Mix1	8	82,096	77.14	13.35	9.51	11,102	16.40	4.93	15.50	10.10	4.01	6.86
	12	85,168	76.81	13.37	9.82	17,003	17.55	5.45	12.24	9.16	3.40	5.45
	16	105,771	76.86	13.25	9.89	20,876	17.96	5.74	10.11	8.16	2.95	4.75
	20	122,748	76.90	13.28	9.82	21,129	18.29	5.88	9.15	7.43	2.74	4.34
	24	155,872	76.83	13.19	9.98	22,544	18.42	5.94	8.49	6.97	2.57	3.94
Mix2	8	85,702	77.28	13.23	9.49	12,124	16.26	4.92	15.42	10.93	4.38	7.52
	12	91,582	76.99	13.22	9.79	17,886	17.04	5.31	12.56	10.71	3.34	6.51
	16	112,150	77.12	13.24	9.64	22,196	17.41	5.38	10.19	10.15	2.80	5.85
	20	131,536	77.10	13.24	9.66	24,574	17.66	5.45	9.95	9.89	2.53	5.57
	24	166,810	77.27	13.20	9.53	27,058	17.77	5.42	9.99	9.91	2.42	5.34

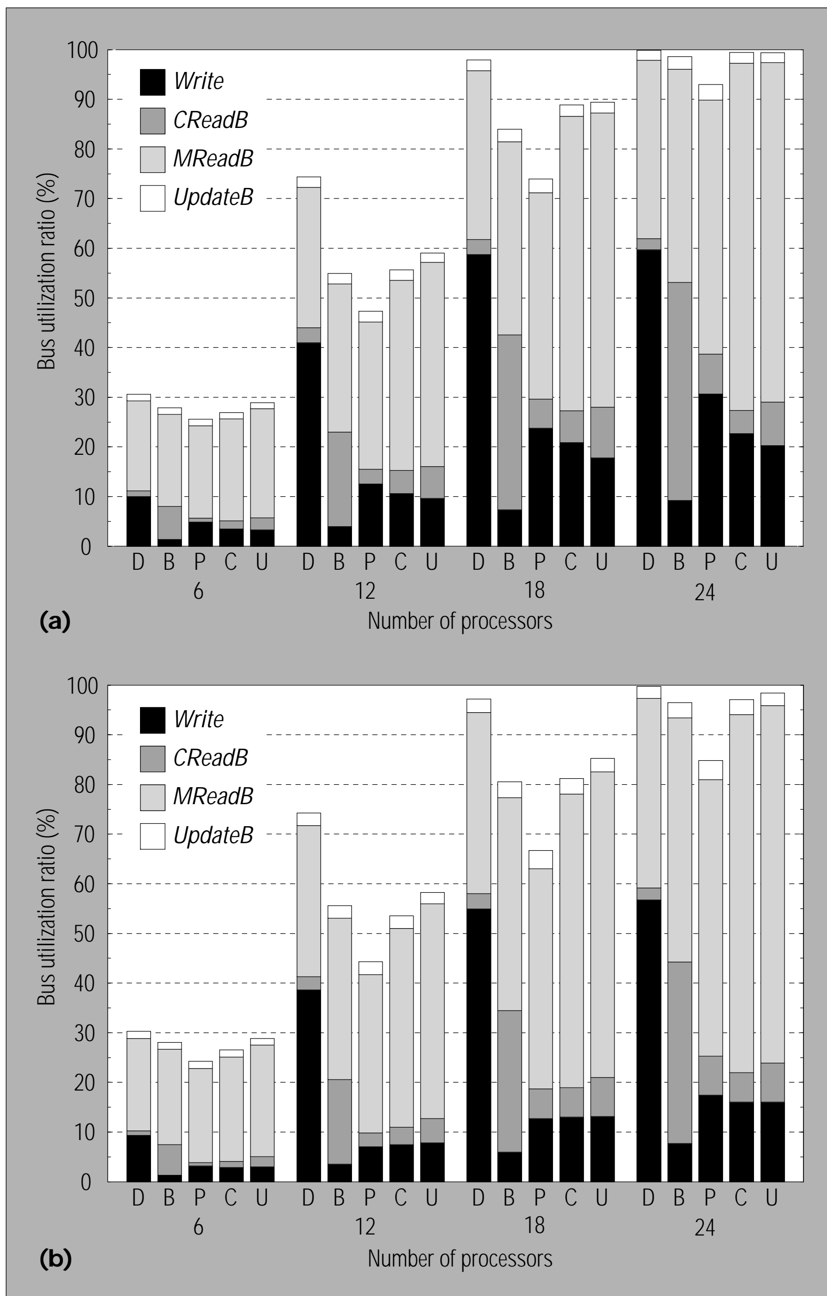

Figure 8. The bus-utilization ratio, segmented by transaction type, for workloads (a) Mix1 and (b) Mix2. For each processor configuration, the figure shows the behavior of the five protocols (D = Dragon, B = Berkeley, P = PSCR, C = Competitive Snoopy Caching, and U = Update Once). The Write bar also includes invalidation transactions for those protocols that use this kind of transaction.

performance for each protocol (except Dragon), because of the active-shared-copy overhead required to keep those copies coherent (see Figure 7a and b). (This effect is also evident on the bus-utilization-ratio graphs in Figure 8.) In the case of Dragon, this behavior is not appreciable, because it saturates the bus, starting from a low number of processors. Even for these workloads, the behavior changes with the block size, and Berkeley exhibits the worst penalization with the 128-byte block. Medium-grain sharing (Mix2) decreases overhead (see Figure 7c and d). Because the protocols adopt different invalidation strategies, they behave differently for different timing costs for write, invalidation, and read-block transactions. In particular, Competitive Snoopy Caching takes advantage of Timing II.

Finally, Table 8 reports the statistics of target traces obtained for Dragon with Timing I. We focus on Dragon to better highlight passive-sharing effects, because Dragon does not use shared-copy invalidation.

Comparing the write-run statistics of Tables 6 and 8 reveals how kernel activities and uniprocess applications (particularly process migration) modify the write-runs of source traces. Because the write-run, or in other words the sharing pattern, influences the cost of maintaining coherence, the introduction of kernel modeling is strongly motivated in the evaluation of such multiprocessors.

This example of performance evaluation shows that Trace Factory can

- produce traces that include aspects of kernel activity, starting from tracing tools that cannot capture kernel references; and
- generate more flexible traces (for example, adding more applications, varying the number of processors, or varying parameters that influence the scheduling or the virtual-memory management), starting from traces obtained with other tools.

We performed all simulations on a PC using a 133-MHz Pentium and the Linux operating system. The estimated total computation time was 32 μs per reference, of which 1.2 μs (3.75%) was required by Trace Factory.

Our methodology is useful in all the situations in which we have a tracing tool that cannot record kernel references. We used Trace Factory to evaluate shared-bus, shared-memory multiprocessors, but it can also work for other models of multiprocessors. Other evaluation methodologies (such as those based on complete simulation) can obviously provide better accuracy, but our proposed solution represents a good trade-off between speed, accuracy, and complexity. In particular, the environment is very flexible, and the evaluation of a machine running special and critical workloads requires short setup and simulation time. Our hybrid methodology will be useful also in other fields in which standard tools eventually cannot trace all types of references. This is the case, for example, with embedded-system evaluation.

Acknowledgments

The Ministry of University and Scientific and Technological Research (MURST), Italy, and the Unviersity of Pisa, Italy, supported this work. 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. We are particularly grateful to Pierfrancesco Foglia for contributing significantly to the validation of our proposed methodology. Our discussions with Veljko Milutinovic and Per Stenström helped improve this article considerably. Finally, we thank the anonymous reviewers for their valuable comments and suggestions.

References

1. K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.
2. M. Tomasevic and V. Milutinovic, *The Cache Coherency Problem in Shared-Memory Multiprocessors—Hardware Solutions*, IEEE Computer Society Press, Los Alamitos, Calif., 1993.
3. S.J. Eggers, "Simulation Analysis of Data Sharing in Shared Memory Multiprocessors," PhD dissertation, UCB/CSD 89/501, Computer Science Dept., Univ. of California, Berkeley, Calif., 1989.
4. 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 Systems*, Vol. 6, No. 9, Sept. 1995, pp. 915–929.
5. C.B. Stunkel, B. Janssens, and W.K. Fuchs, "Address Tracing for Parallel Machines," *Computer*, Vol. 24, No. 1, Jan. 1991, pp. 31–45.
6. M.A. Hollyday and C.S. Ellis, "Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulation," *IEEE Trans. Parallel and Distributed Systems*, Vol. 3, No. 1, Jan. 1992, pp. 97–109.
7. C. Prete et al., "Some Considerations about Passive Sharing in Shared-Memory Multiprocessors," *IEEE TCCA Newsletter*, Mar. 1997, pp. 34–40; http://computer.org/tab/tcca/news/mar97/prete.pdf.
8. M.S. Squillante and E.D. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 2, Feb. 1993, pp. 131–143.
9. B. Vashaw, "Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors," Tech. Report CMUCDS-93-4, Dept. of Electrical and Computer Eng., Carnegie Mellon Univ., Pittsburgh, 1993.
10. C.A. Prete, G. Prina, and L. Ricciardi, "A Selective Invalidation Strategy for Cache Coherence," *Inst. Electronics, Information, and Communications Engineers (IEICE) Trans. Information and Systems*, Vol. E78-D, Oct. 1995, pp. 1316–1320.
11. J.G. Gee and A.J. Smith, "Evaluation of Cache Consistency Algorithm Performance," *Proc. MASCOTS '96: Fourth Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, IEEE CS Press, 1996, pp. 236–248.

Roberto Giorgi is a PhD student in computer engineering at the University of Pisa, Italy. His interests involve computer architecture themes such as coherence protocols for multiprocessors, the behavior of user and system code, architectural simulation, and multithreaded processors. He took part in the ChARM project in cooperation with VLSI Technology Inc., San Jose, California, developing part of the software used for performance evaluation of ARM-processor-based embedded systems with cache memory. He received his MS in electronic engineering, summa cum laude, from the University of Pisa, with a thesis on multiprocessor trace-driven performance evaluation. He is a member of the IEEE, IEEE Computer Society, and ACM. Contact him at Dip. Ingegneria della Informazione, Univ. di Pisa, via Diotisalvi 2, I-56126 Pisa, Italy; giorgi@acm.org; http://www.iet.unipi.it/~giorgi/.

Cosimo Antonio Prete is an associate professor of computer systems at the Department of Electronic, Computer, and Telecommunication Engineering at the University of Pisa, Italy. His research interests include multiprocessor architectures, cache memories, and performance evaluation. He has performed research in debugging environments for distributed systems, commit protocols for distributed transactions, cache-memory architecture, coherence protocols for tightly coupled multiprocessor systems, and software environments for teaching computer architecture. He has been project manager for the University of Pisa for the Esprit III Tracs project (a flexible real-time environment for traffic control systems) and for the Cache-Sim project (a framework for the modeling and simulation of cache memories in ARM-based systems), which produced ChARM for VLSI Technology Inc., San Jose, California. He has also acted as an expert on the Open Microprocessor Systems Initiative for the Commission of the European Communities. He earned his undergraduate degree in electronic engineering, summa cum laude, and his PhD in computer engineering from the University of Pisa. He is a member of the IEEE, IEEE Computer Society, and ACM. Contact him at Dip. Ingegneria della Informazione, Univ. di Pisa, via Diotisalvi 2, I-56126 Pisa, Italy; prete@iet.unipi.it; http://www.iet.unipi.it/~prete/.

Gianpaolo Prina has performed research in coherence protocols for tightly coupled multiprocessor systems. He is a consultant at the Department of Electronic, Computer, and Telecommunication Engineering at the University of Pisa. His research interests include cache memories, multiprocessor architectures, and trace-driven simulation. He received his degree in electronic engineering, summa cum laude, from the University of Pisa, and his PhD from the Scuola Superiore S. Anna in Pisa. Contact him at Dip. Ingegneria della Informazione, Univ. di Pisa, via Diotisalvi 2, I-56126 Pisa, Italy; prina@iet.unipi.it.

Luigi Ricciardi is a software engineer at Intecs Sistemi SpA, Pisa, Italy. He has performed research in coherence protocols for tightly coupled multiprocessor systems. His research interests include cache memories, multiprocessor architectures, and trace-driven simulation. He received his degree in electronic engineering from the University of Pisa in 1992. Contact him at Intecs Systemi SpA, Via Gereschi, 32/34, 56127 Pisa, Italy; ricciard@iet.unipi.it.