Limes: A Multiprocessor Simulation Environment for PC Platforms

Igor Ikodinovic, Aleksandar Milenkovic, Veljko Milutinovic [a]
Davor Magdic [b]

[a]Department of Computer Engineering,
Faculty of Electrical Engineering,
University of Belgrade,
Bulevar Kralja Aleksandra 73, 11000 Belgrade, Serbia, Yugoslavia,
igi@eunet.yu, emilenka@etf.bg.ac.yu, vm@etf.bg.ac.yu

[b]Department of Computer Science,
University of California, Santa Barbara,
Santa Barbara, CA 93106,
davor@cs.ucsb.edu

Abstract

This paper presents a multiprocessor simulation environment, developed with the aim to facilitate the researches of multiprocessor systems using widely available hardware platforms. It comprises simulation tools, including both an execution-driven and a trace-driven simulator, applicable for memory architecture studies of shared-address space multiprocessors. It also includes a detailed model of a bus-based cache coherent symmetrical multiprocessor system. The execution-driven simulator can run parallel applications based on the ANL programming paradigm, such as those found in the SPLASH-2 suite, and employs a scheduling algorithm specially optimized for speed. The trace-driven simulator is based on a concept of ideal traces, and supports an original technique for abstraction of events that can introduce timing dependencies in a trace, which in turn enables accurate simulation. Both the execution-driven and the trace-driven simulator can work using same simulators of memory architectures. The environment provides a simple general interface that allows for the hardware lying underneath the simulated processors to be modeled using object oriented programming. The package currently runs on PC platforms with Pentium or newer processor under Linux operating system.

1 Introduction

Simulation plays a vital role in multiprocessor studies. In a variety of simulation techniques, ranging from analytical modeling, that is often inadequate for being unable to model complex multiprocessor interactions, to hardware prototyping that is costly and inflexible, software simulation has become very popular. Software simulation has

certain benefits that make it the dominant method for validating of proposed architectures. Software simulators are easier to develop, they are less expensive than their hardware counterparts and they are able to perform simulations with high level of accuracy. They are also more flexible, allowing frequent changes of simulation parameters and easy changes in the simulated architecture; this is significant because details of the architecture under evaluation may frequently need readjusting, according to the simulation results.

The world of software simulation comprehends several different simulation methods, with a number of tools that follow them [1]. Certain trade-off between accuracy, speed, flexibility, expense, portability, and ease of use is present in every simulation method. These issues should be carefully considered when evaluating simulation techniques or comparing them with each other. A rapid development in the computer field can also change conditions that make some method the best at one point of time, leading toward the introduction of new methods or toward reemerging of some of the old ideas, so that occasional reevaluation of the simulation techniques is necessary.

Currently, one of the most popular methods for simulation of multiprocessors [2] is execution-driven simulation, due to its speed and accuracy. This method has been used in a number of popular simulation tools. However, this method does not enable OS references to be included in the simulation. Trace-driven simulation[1] is another method that was widely used in the past, but was replaced by other techniques, mostly due to its need for large disk space, problems with low disk transfer rates, and inability to accurately simulate complex interprocess interactions. Yet, with rapid development of technology, disks with capacities of 10GB or more and with transfer rates over 10MB/s have become widely accessible, eliminating some of the drawbacks of this method. Trace-driven simulation can be performed with traces that can contain memory references from any source, including those from OS. Limes benefits from using both of these two simulation methods.

Limes consists of two simulators (execution-driven and trace-driven simulator) and a modifiable software representation of a realistic multiprocessor system. It currently runs on PC platforms with Pentium or newer processor under Linux operating system.

The following sections will describe Limes. In section 2 we explain the goals authors sought to satisfy with Limes and other requirements set before it. In section 3 we discuss the existing simulation tools and why is Limes different from them. Section 4 presents Limes structure in details, including the execution-driven and the trace-driven simulator and their internal algorithms, and gives insight in one realistic memory architecture model on an example of the SMP system. Section 5 speaks of Limes complexity and performance. We draw the conclusions in section 6.

[1]The term trace-driven simulation sometimes pertains to all types of software simulations, meaning that a stream of memory references constitutes a trace no matter where it comes from (from direct execution or from a file). Here, by execution-driven simulation we mean that memory references come from direct exection of the instrumented code, and by trace-driven simulation that memory references come from a file residing on a disk.

2 Goals and demands

The nature of researches that were to be performed, mainly involving shared-memory multiprocessor studies, imposed a specific set of demands for a simulation environment that would be considered appropriate.

First, it had to be executable on PC platforms, because of the estimated number of experiments, the availability of such platforms in the environment, and the general inaccessibility and lower number of high-end machines. Still, we wanted to keep a possibility of porting the environment to other platforms by need.

Second, simulations were meant to be driven by realistic workload - the kind of workload that would execute on the proposed hardware platform once it comes to life. Character of the workload greatly influences the performance indicators, and using inadequate workload can often be misleading. For the execution-driven simulator we had to choose an appropriate set of applications it can run, in order to give the simulation results the necessary validity. The SPLASH-2 application suite [3] was considered a preferred workload, since this set of benchmarks became a de facto standard among the researchers involved in multiprocessor studies. The trace-driven simulator was meant to be able to work with traces generated in our environment, or elsewhere, by need. Having both of these two types of simulators would enable simulations with a wide range of different workloads.

Third, the simulators were to deliver high level of accuracy, having in mind the character of the studies, like simulations of bus-based cache coherence protocols [4].

Memory architectures that are studied often have certain similarities in their structure, and the subtle differences that exist in their internal organization dictate not only that the simulation be precise, in order to accurately measure the impact of these differences on performance, but also that the simulator can be easily changed and adapted, so that little effort is spent when changing details in their structure. These and other requirements suggested the use of object oriented programming (OOP) techniques in the building of the memory simulators. Writing a simulator using an OO language (such as C++) allows great freedom to the writer of the simulator. The OO approach is also good when considering the desired level of efficiency. Having in mind the number and the volume of the experiments that need to be performed and a need for frequent changes of simulation parameters and memory architecture details, the advantages of the OOP approach become fully visible.

Certain other conditions had to be fulfilled, such as the need to develop memory architecture models independently from the simulation kernel, which would give the opportunity for development and testing of systems using parallel traces that come from different sources (i.e., from direct execution, or from a trace residing on a disk). A part of the simulator that handles the instrumentation of the application assembly code (for execution-driven simulation) was also meant to be written to be independent, which would allow us a possibility to change it easily when switching to different platforms and thus requiring no other changes in the rest of the environment in that process.

3 Existing solutions

Vast majority of the existing simulation tools is developed either for high-end multiprocessor computers or, if made for uniprocessor machines, then those are almost exclusively platforms with RISC processors (mainly MIPSes and SPARCs). Sophisticated tools like SimOS [6] (runs on a MIPS based SGI multiprocessor) and SimICS [7] (runs on SPARC machines) can simulate target architectures with high level of accuracy using instruction set emulation. They are able to simulate the execution of an entire realistic operating system on a target machine, including complete simulation of the I/O subsystem. Beside the operating system itself, all kinds of applications can be used as a realistic workload for the simulations as well. One thing that doesn't allow these tools to be considered perfect, except that they work only on RISC platforms, is the speed of simulation, that is still lower than in the case of the execution-driven simulation.

Tools like Tango [8] and its successor TangoLite [9], or CacheMire [10], are widely used execution-driven simulators; but again, they only work on RISC platforms. They can not ensure a satisfying level of accuracy if ported to a different hardware platform, such as a CISC uniprocessor. Augmint [11] is the only such tool that does in fact run on a PC (with a minor drawback of omitting iAPx86 string instructions and standard library routines from instrumentation). Augmint has an advantage that it also runs under Windows NT, as well as under Solaris operating system on SPARC machines. However, a trace-driven simulator is not included in the Augmint environment, making it virtually impossible for OS references to be used as a workload in the simulations. TangoLite and CacheMire are also lacking a possibility of trace-driven simulation. They can only produce traces.

Having in mind insufficiencies of the existing tools regarding trace-driven simulation, nonexistence of such tools for PC platforms and for CISCs in general (Augmint was also just being developed at the time), and an awareness that every tool almost invariably requires modifications in order to allow for particular effects to be simulated and measuring techniques to be added, we felt that it would be worth of effort to invest the time in developing a simulator that would be well suited for our research goals, rather than to modify the existing tools. Once we decided to put effort to it, we set as a goal to develop the whole environment as a general tool for simulations of all shared-memory multiprocessor architectures.

4 Limes structure

Limes simulation environment comprises both an execution-driven and a trace-driven simulator. They enable a multiprocessor system to be simulated on a uniprocessor host machine, in our case a PC based on Pentium or newer processor. The simulator of the target system[2] is devised to be independent from the source of memory references.

[2]The simulator of the target system actually simulates the behaviour of the target memory system. Memory system, for example, can include caches, TLBs, interconnection networks or global memory. Simulation of other parts of the system (like I/O) is not considered here. These are the reasons why this simulator is also called a simulator of the memory system.

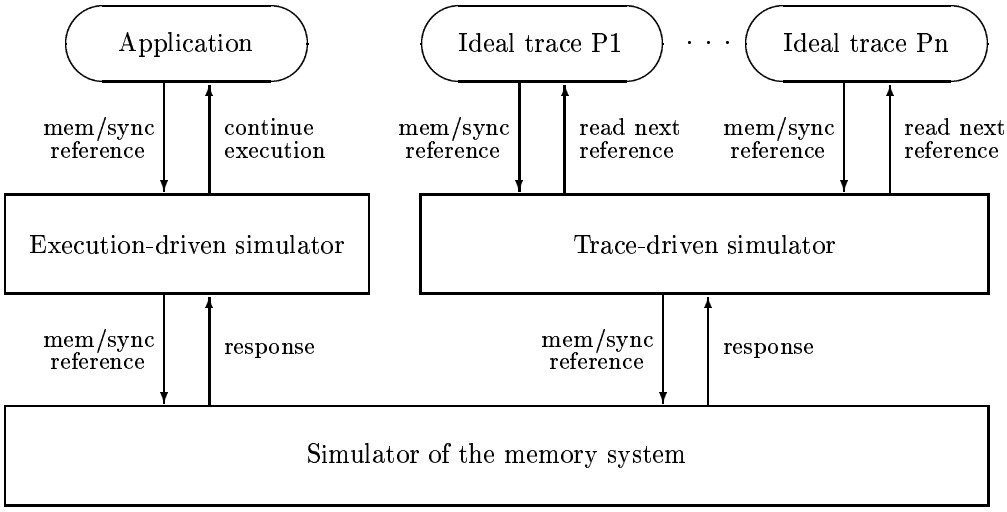

Figure 1: Simulation in the Limes environment

This enables that both the execution-driven and the trace-driven simulator can use same simulators of the memory system, where memory/synchronization references come either from the execution streams of the parallel application's threads or from the traces residing on a disk, depending on the type of the simulation. This is shown in Figure 1.

The environment also includes an extension of the ANL programming paradigm, where parallel programs are written using only two classes - threads and monitors. This programming paradigm is easier to use and programs written in this manner are more understandable than those using raw LOCKs and UNLOCKs. The whole system is rather small, built on top of Limes. Other programming paradigms can be implemented in a similar way. This can be of further value for researchers who may not be necessarily concerned with architecture studies, but are primarily focused on investigating parallel algorithms.

4.1 Execution-driven simulator

Execution-driven simulation is comprised of two main processes: building of the simulation executable, which is performed only once, and simulated execution of the parallel application, which can be performed repeatedly once the executable has been built.

4.1.1 Building an executable

Applications that can be used to drive the simulator are like those from the SPLASH-2 suite, i.e, they must use the ANL (Argonne National Lab) macros for expressing parallelism. When compiling the application, ANL macros are first expanded into the

series of C/C++ instructions, and then, after the compilation, the whole application assembly code is instrumented with call-outs of the simulation kernel. The simulator allows three levels of instrumentation, depending on what type of events are to be simulated. Level 0 instruments only synchronization primitives and user defined machine instructions (which are also possible to define in Limes - for example prefetch, forward etc.), level 1 instruments shared reads and writes as well, and level 2 instruments all other memory references also, including local ones. The speed of simulation is reciprocal to the instrumentation level. Level 0 may be appropriate for investigation of parallel algorithms, level 1 for shared-memory architecture evaluations, and level 2 for simulations of caches.

Parallel applications, like most other C/C++ programs, call standard library functions. Limes by default instruments all standard library functions whose argument can be a pointer, as they might read or write global memory without the control of the simulator. Other library functions are not instrumented (which has little impact on the accuracy), but can be if needed; instrumentation tool is open for additions.

What makes the whole instrumentation process hard is that CISC (unlike RISC)[3] instruction set is very complex, containing many instructions, that are frequently non-uniform, so that many addressing combinations are possible.

The instrumented application code, the simulator kernel, and the simulator of the memory system are finally compiled and linked together into a single executable. If the application code is compiled with an appropriate option, the compiler can produce some debugging information that the instrumentation tool can understand. If the application crashes, the simulator will use these pieces of information and print the source line that the offending thread was executing at that moment. The instrumentation does not prevent the application to be debugged with a standard debugger.

4.1.2 Simulation

Basically, during the execution the simulator kernel acts like a layer between the parallel application and the simulated memory system. Parallel application executes its native machine code, one thread at a time, until it encounters an event of interest, such as a memory/synchronization reference (read/write, lock acquire/release, or some user defined instruction). Then it gets rescheduled, and the actual operation

[3]One important issue deserves attention here: knowing that the simulator will execute applications that are compiled for a CISC processor, the question is whether the simulation results are bound to the behaviour of such a processor. Can it be used for simulating some future RISC multiprocessor too? It is hard to give an exact answer, but our results show that it is possible. One real SPLASH-2 application (FFT) was executed on a MIPS R2000 DEC station and one on a Pentium based PC, with same parameters (65536 complex doubles - a realistic problem size). The first simulation was performed using TangoLite, and the second one using Limes. MIPS generated 18.393 millions of shared reads and 12.908 millions of shared writes, while Pentium generated 18.552 millions of shared reads and 12.782 of shared writes during the execution of the application. The results indicate that for a real application, a RISC such as MIPS R2000 generates approximately the same number of shared reads and writes as Pentium does (in both cases the discrepance is less then 1references are generally of little importance to multiprocessor studies; shared references is what determines the behaviour and performance of multiprocesor systems.

is deferred until its time stamp reaches the global simulation time. The simulator is responsible for multiplexing threads and for scheduling of their memory requests at proper times, in correct order.

Context switching occurs each time the thread is rescheduled. The programming model supported is lightweight threads model (all processes share the address space with the master process, except for the stack area, which is private for each process). Consequently, the whole simulation (including application threads, simulator kernel, and memory simulator) executes in the context of a single UNIX process.

Instruction execution times are calculated at the end of each basic block (a basic block here ends with a branch, a label, or a memory reference). At compile time each instruction is associated with a number that represents the number of cycles it takes to execute before the execution of the next instruction begins. For most instructions this number is one. By changing these values it is possible to model a faster or a slower processor. For example, by decreasing their value we can roughly simulate a superpipeline processor. Simulation time is expressed in processor cycles of the target multiprocessor system.

The simulator can also produce a trace that contains references according to the level of instrumentation, along with additional data that can be used to support accurate trace-driven simulation; file format is open for the user to change it.

4.1.3 Optimized scheduling algorithm

Scheduling is necessary during the execution of the simulation to maintain correct interleaving of memory activities of the application threads. The scheduling algorithm of the simulator can substantially influence the performance of the simulation. Though relative importance of the scheduling overhead on the simulation performance is limited by the existence of overhead introduced by other factors (primarily by the code instrumentation and by the simulation of the memory system), the frequency of scheduling activities make it worth-wile to reduce the scheduling overhead as much as possible. The following optimizations are implemented:

1. Accurate simulation can be accomplished if the memory simulator was called after every cycle in the simulation, regardless of whether any requests exist in that cycle or not. However, it would slow down the simulation significantly. Another, equally correct approach is chosen: the scheduler calls the memory simulator in subsequent simulated cycles only if there is at least one new request or at least one stalled thread; idle cycles can be skipped.
2. The scheduling algorithm will force the continuous execution of non-global instructions as much as possible, saving thus the time needed for frequent invocations of the scheduler and context switchings.
3. The scheduler will call the memory simulator only if there are no threads that can continue execution, i.e. if they all wait for a memory operation to complete, saving thus additional time for invocations of the memory simulator and context switchings.
4. Threads are allowed to continue execution without first waiting for shared writes to be completely simulated, deferring that job as long as possible and saving thus the time for frequent calls to the memory simulator.

4.2 Trace-driven simulator

Making a trace of some program's execution enables that a simulation with the same workload can be performed without a program's reexecution. Trace-driven simulation is concerned only with the global memory references and synchronization operations, while local operations are not of importance for multiprocessor studies (their influence is reduced to the time of their completion and it is built in a trace via time stamps).

There are some benefits of using trace-driven simulation. If multiple simulations are performed with the same trace then trace-driven simulation can potentially bring speedup over multiple execution-driven simulation runs; unfortunately, due to the disk transfer rate limitations, this is not always possible. Multiple runs can also yield different results when using execution-driven simulation if non-deterministic scheduling is performed. It is possible even with static scheduling of workload. This is not the case if trace-driven simulation is used. Trace-driven simulation also enables the use of traces that contain OS references, while execution-driven simulations can not include OS references at all.

There is one more benefit from using traces as a workload. When comparison of two architectures is needed, and their simulators are not working on identical platforms, or even with the same simulators, trace-driven simulation enables completely accurate comparison (only instruction interpretation method enables that too). Using execution-driven simulation, for example, could give completely different results for the same workload. Same traces, on the other hand, contain same information on the local operations (i.e., they have same time-stamps), so the comparison is reduced only to differences in the memory systems, whose simulation is independent of the platform.

Limes' trace-driven simulator can use as input a trace generated in the Limes environment or from some other source. Limes is very flexible concerning the trace file format. It currently supports two formats: a textual trace file format that enables a trace to be viewed with a simple text editor, and a binary trace file format that is significantly shorter and used for storing large traces. Other formats can be easily implemented by altering the appropriate modules of the simulator. Every format, however, should be able to support abstraction by allowing some additional information to be stored in a trace; Limes enables accurate trace-driven simulation, where possible, by using the method of abstraction to eliminate timing dependencies.

4.2.1 Accuracy issues

The execution path of a multiprocessor workload depends on the ordering of events in a system, which in turn depends on the timings of the machine's memory system. When timing dependencies are present, a small change to the memory system architecture can induce numerous changes to the execution path of a program and cause inaccurate simulation. This happens because the only information that a trace contains about the execution path of a program is the one implicitly built in a trace via time stamps, which is valid only until the memory system is not changed. If the memory system is changed, timings may get changed too, and, consequently, program's execution path can be changed.

In [12] authors find that traditional address traces are not adequate for accurate trace-driven simulation and they propose intrinsic traces as an alternative (intrinsic trace consists of the control-flow graph of the workload plus timing and address data for each basic block). They argue that accurate trace-driven simulation without partial reexecution of the program is possible only for so called graph-traceable programs (those are programs where all addresses can be determined during simulation, based only on the information gathered from the trace). In reality the range of programs for which accurate trace-driven simulation can be obtained at a reasonable cost reduces to a class of programs whose threads have execution and data paths that can not be influenced by other threads. Most applications comply to this condition.

As discussed in [13], accurate trace-driven simulation can be obtained (where possible) only by eliminating timing dependencies from the trace by abstraction of the operations that cause them. To eliminate timing dependencies by means of abstraction, we must first ensure that all the necessary information are recorded in a trace, and then to support the abstraction on the side of the simulator (at the expense of additional simulator complexity).

In [14] it is discussed whether traces generated from multiple runs of the same program will yield the same results, and if tracing induced dilation affects simulation accuracy. As already mentioned, we don't need multiple simulation runs, because once we get a trace it can be used for simulation of different memory architectures. As for the second issue, Limes produces traces without the time dilation effect, but we can not guarantee that trace collection techniques used elsewhere [15] will not introduce that effect. For example, like Limes, TangoLite, Augmint, and other execution-driven simulators that are used for trace collection, do not introduce time dilation. Traces generated by SimOS or SimICS also do not suffer from the time dilation effect. This is because they all perform tracing on the simulated architectures. Microcode modification used in ATUM [16] and a technique of inline tracing used in MPtrace [17] and TRAPEDS [18] does introduce time dilation, as they are used to trace programs on a host machine. The dilation effect, however, is often negligible.

4.2.2 Abstraction

Abstraction of operations that can influence program's execution path means that certain information about them are stored in a trace so that they can be correctly redone during the simulation. That way the simulator will no longer be bound to the time stamps when the execution path is concerned; instead, it will be able to maintain the correct ordering by redoing the critical timing dependent operations. Timing dependent operations include: shared memory references (reads and writes), synchronization operations (such as locks and barriers), operations for dynamic scheduling of workloads (like allocation of tasks, loop iterations, or memory), and other timing dependent operations (like creation of a child thread). Timing dependent operations other than those that can influence the program execution path (like those involving real-time clock[4]) can also be abstracted.

[4]Operations involving real-time clock are timing dependent because the timings of other operations influence the time that the clock shows at certain point.

We divide timing dependent operations in two groups: elementary and complex. Elementary operations are those that are realized in the simulated hardware as primitives, or represent simple events whose influence on the simulation can be completely defined by their timing information. Complex operations are realized through the use of elementary operations. In our case elementary operations are shared reads/writes, lock acquires/releases, user defined instructions, and events of creating a child thread, while barriers are realized as complex operations; however, it can be changed which operations will be elementary and which will be complex. The simulator is also ready to support the abstraction of other timing dependent operations and of different implementations of currently supported operations.

Elementary and complex operations are abstracted in a different way. Timing dependencies can be eliminated from elementary timing dependent operations by recording their timing information (start, end, or duration). That way the simulator can calculate the amount of 'pollution' introduced by the architecture on which the trace was generated and eliminate it. Complex operations are much harder to abstract, depending on how "complex" they are. In general, it is up to the trace-collecting tool to save all the necessary information into the trace. The abstraction technique we use is responsible to perform accurate simulation based on them.

4.2.3 Concept of an ideal trace

The abstraction of elementary timing dependent operations can be reduced to simply decreasing the time stamps for the amount of their time of duration. In another words, time stamps, that implicitly contain the information about the execution path, are reduced to contain *only* that information and *no* information on the memory system. This needs to be done only once after the trace is created, and after that the simulations can be performed any number of times. Complex timing dependent operations, on the other hand, must be handled completely by the simulator at run time.

We based our simulator on ideal traces. Ideal trace is a trace where all elementary timing dependent operations have already been freed of timing dependencies by adjusting their time stamp values, and where complex timing dependent operations have been properly abstracted. The easiest way to produce it is to run a simulation using an ideal memory simulator, where each memory request is completed in a single cycle. However, lock acquire operations will not be abstracted that way (for they must preserve correct global ordering). We fight this inconvenience by forcing the first and the last attempt for gaining a lock to appear in a trace. After the simulation is over, during the postprocessing phase, the superfluous lock appearances are eliminated, while the time stamps of all the references that follow are corrected for the amount of time the lock had to wait to gain it. This procedure assures a fast way to get an ideal trace, as ideal memory simulator adds very little time to the simulation overhead.

4.2.4 Simulation using Tracer

Tracer is Limes' trace-driven simulation tool. By default, Tracer supposes that a trace it will use as an input is a single file that contains all processors' references. This format is chosen because it is more efficient regarding storage space than having a separate trace file for each processor. It allows that one time stamp can be used for a group of operations that are done in the same cycle on different processors. It is more efficient to have same number of 1-byte processor labels than 4-byte time stamp integers. Such a trace is also convenient for visual inspection if it is in textual format. However, this format is not convenient for the simulation. That is why Tracer first invokes a parser that makes ideal traces for each processor, which are then used as an input to the simulator (see Figure 1). At the same time parser eliminates superfluous lock requests and corrects time stamps (as a part of the postprocessing phase of the lock abstraction process), and extracts in a special data structure information needed by the simulator to support abstraction of barriers. This process needs to be done only once, and the obtained ideal traces can be then used for multiple simulations. This is much more efficient than if the whole procedure was done by the simulator at run time. That also means that no additional information on the elementary timing dependent operations are needed in a trace after postprocessing, which reduces its size.

The simulation process is quite straightforward, except for the barriers. In a complex implementation where each barrier in a trace is just a set of reads, writes, locks and unlocks, there would be no way to tell if these primitives functionally execute this synchronization operation if they weren't previously annotated during the trace generation phase. Each barrier is annotated with markers at certain points, which are recognized by the scheduler, and by using these pieces of information it can perform the abstraction. The abstraction principle is quite general and can be used for handling different implementations of barriers, as well as other complex timing dependent operations. Another event that must be recorded and abstracted is the creation of a child thread. It is necessary to abstract these events, to prevent scheduling of child threads before they were actually created by the parent.

This set of abstracted operations is sufficient to support all statically scheduled workloads. Abstraction of dynamically scheduled workloads can be supported by extending the abstraction paradigm used for statically scheduled programs.

4.3 Memory system simulator

The simulator of the memory system can be used by both the execution-driven and the trace-driven simulator. The simulator of the memory system is completely independent from the simulation kernel and from the applications (traces) used to drive the simulations. The simulation kernel provides the simulator of the memory system with a stream of requests. They use a simple interface to communicate, and that interface should be used when building up a simulator of any target system. How is the memory simulator realized is of no importance as long as it uses that interface.

4.3.1 SMP memory simulator

A model of a bus-based cache coherent symmetrical multiprocessor system (SMP) comes with Limes. It is a system comprised of N identical processors, with on-chip (L1) caches implementing one from the set of supported coherency protocols. The processors and the main memory are interconnected via bus, and all the communication through bus signals is adequately mimicked.

The system currently includes detailed examples of five snoopy cache coherence protocols, namely WTI (Write-Through Invalidate), WIN (Word Invalidate), Berkeley, Dragon, and MESI. The protocols differ in the operation of their cache controller modules, while the rest of the modules are functionally equal. The operation of the module is distinctly represented as a mixture of a flow chart and a state diagram. Modules are easily modifiable and extendible since they are written employing the OOP style.

Like cache controllers, all other hardware units are also programmatically represented as independent modules (C++ classes). That is, the modules do not communicate directly; rather, they are organized as isolated units, which communicate with the outer world through input/output ports: a module reads its input ports, performs the operation that depends on both the information on the input ports and the internal state of the module, and leaves the result on its output ports. This design philosophy resembles greatly the one employed by VHDL. This only shows that various approaches are possible using OOP; the freedom in writing a memory system simulator, however, should be complete. A guide to designing simulators is a part of the Limes documentation.

5 Limes complexity and performance

The whole Limes package, including both simulators and all memory models, consists of around 8500 lines of C++ code. The most complex classes - the detailed cache controllers, contain on the average some 400 lines that capture the complete behavior of the controller. The whole code is profusely commented. Complete Limes environment (including 9 SPLASH-2 applications) takes about 820KB, compressed.

The compilation process is rather quick, and takes from 1s. for simple memory models, up to 15s. for the most complex ones, measured on a Pentium/133 platform.

Limes performance is presented in Table 1. The results show execution times and slowdowns of four SPLASH-2 programs, for execution-driven simulations. Simulations have been done for 3 different memory models and 2 instrumentation levels for each model. Abstract model does not invoke the scheduler or the memory simulator. It responds to the requests right away, but still preserves the global ordering. Ideal and MESI models both invoke the scheduler and the memory simulator. Ideal memory simulator returns a satisfy signal in a single cycle for every memory request (read/write), except for synchronization requests (lock/unlock). MESI is the most complex memory simulator in the current version of Limes, performing the simulation of a bus-based SMP with MESI cache coherence protocol.

simulator/ level	OCEAN		FFT		LU		RADIX		Avg. slw.
	time[s]	slw.	time[s]	slw.	time[s]	slw.	time[s]	slw.	
uninstrumented	8	1	2	1	16	1	7	1	1
abstract/level1	312	39	65	32	863	54	77	11	34
abstract/level2	684	85	94	47	1120	70	270	38	60
ideal/level1	729	91	122	61	2438	152	132	19	81
ideal/level2	1334	166	200	100	3045	190	698	100	139
MESI/level1	5273	660	850	425	10056	628	737	105	454
MESI/level2	5694	712	893	447	10297	644	1798	256	515

Table 1: Execution-driven simulation times and slowdowns for 4 SPLASH-2 applications, for various memory simulation models. Simulations are for 16 processors. They were ran on a Pentium/133MHz PC platform.

simulator/ type	OCEAN		FFT		LU		RADIX		Avg. ratio
	time[s]	ratio	time[s]	ratio	time[s]	ratio	time[s]	ratio	
MESI/trace	145	2.38	57	2.37	36	1.80	66	1.94	2.12
MESI/exec	61		24		20		34		

Table 2: Speed of the trace-driven simulation for 4 SPLASH-2 applications compared to the execution-driven simulation. Simulations are for 8 processors. They were ran on a Pentium/133MHz PC platform.

Slowdowns for abstract model indicate the instrumentation overhead introduced by the simulator, where correct global ordering is still kept. Results show that this is the biggest source of slowdown compared to other factors. Slowdowns using ideal memory simulator indicate the scheduling and the memory simulator invocation overhead. They are relatively low due to the optimized scheduling policy. Finally, simulation of a realistic bus-based SMP system with a MESI protocol indicates the memory simulator overhead.

The results are comparable to the TangoLite performance. Limes has an average slowdown of 454 compared to the TangoLite average slowdown of 765 for the similar simulation complexity.

Results for the trace-driven simulation are presented in Table 2. They show the trace-driven simulator performance against the execution-driven simulator. Disk transfer rate is about 1.0 MB/s.

It is obvious that disk transfer introduces a constant overhead in the simulation. It can be substantially reduced by using a faster disk. The rest of the time is spent by the simulator. The version of the trace-driven simulator used to obtain these results is not optimized for fast disk access. Trace-driven simulation can be potentially twice faster than the execution-driven simulation with optimal disk access policy.

Instrumentation inevitably increases the size of the application. With Limes, instrumentation typically increases application static size by a factor of 2.1-2.3, which is better compared to the factor of four for TangoLite.

6 Conclusions

Limes comprises two usable simulators and a complete model of an SMP system, offering fast and accurate simulation on today so popular PC platforms. It employs some new abstraction techniques for accurate trace-driven simulation, with a concept that can be extended even to non-deterministic workloads, if properly supported by the trace generation tool. On the other side, the execution-driven simulator offers respectable speed using fully optimized scheduling algorithm. For those that are more interested in investigating parallel algorithms, Limes offers very fast type of simulation that still preserves correct global ordering; also, a new paradigm for easy and comprehensible parallel programming is available, and new ones can be developed.

There is, however, enough room for some improvements and future work. Trace-driven simulator should optimize its access to the trace references and achieve higher simulation speed. It can also be extended to support abstraction of some other types of timing dependent operations, including dynamically scheduled workloads. Simulators could be improved to support multithreading, or thread migration. Future versions of the package may also include simulators of systems other than a bus-based SMP. Development of other types of systems is in progress. In the end, Limes could be ported to work on other platforms and operating systems.

The intentions behind the development of this tool were to facilitate the multiprocessor studies at the University of Belgrade, and to provide the researchers with the environment that can be easily adapted to fulfill their particular demands in order to make their study more effective. The tool can be of benefit to all who need realistic simulations of shared-address space multiprocessors and to the researchers in the field of parallel algorithms. Application of Limes can also be in education, as a free, easily available, understandable, and modifiable tool. The whole Limes package is in the public domain, and can be found at the following Internet address: http://galeb.etf.bg.ac.yu/~vm/smp

References

[1] R. A. Uhlig, T. N. Mudge, Trace-Driven Memory Simulation: A Survey. ACM Computing Surveys, Vol. 29, No. 2, June 1997

[2] S. Goldschmidt, Simulation of Multiprocessors: Accuracy and Performance, Ph.D. Thesis, Stanford University, USA, June 1993

[3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, in *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 24-36

[4] M. Tomasevic, V. Milutinovic, A Simulation Study of Snoopy Cache Coherence Protocols, in *Proceedings of the Hawaii International Conference on System Sciences*, Koloa, Hawaii, USA, January 1992, pp. 426-436

[5] V. Milutinovic, Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lessons Learned, IEEE Computer Society Press, Los Alamitos, California, USA, 1999 (in preparation)

[6] M. Rosenblum, S. A. Herrod, E. Witchel, A. Gupta, Complete Computer System Simulation: The SimOS Approach, IEEE Parallel and Distributed Technology, Fall 1995, pp 34-43

[7] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrm, B. Werner, SimICS/sun4m: A Virtual Workstation, in *Usenix Annual Technical Conference*, New Orleans, Lousiana, June 15-18, 1998

[8] H. Davis, S. R. Goldschmidt, J. Hennessy, Tango: A multiprocessor simulation and tracing system, Technical Report CSL-TR-90-439, Stanford University Computer Systems Laboratory, July 1990

[9] S. A. Herrod, TangoLite: Introduction and User's Guide, Technical report, Stanford University, Stanford, USA, November 1993

[10] M. Brorrson, F. Dahlgren, H. Nilsson, P. Stenstrom, The CacheMire Test Bench - A Flexible and Effective Approach for Simulation of Multiprocessors, in *Proceedings on the 26th Annual Simulation Simposium*, Arlington, USA, March 1993, pp. 41-49

[11] A. Sharma, A. T. Nguyen, J. Torellas, Augmint: A Multiprocessor Simulation Environment for Intel x86 Architectures, CSRD technical report 1463, University of Illinois at Urbana-Champaign, Urbana, USA, March 1996

[12] M. A. Holliday, C. S. Ellis, Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulation, Duke University, Technical Report CS-1990-08, July 1990

[13] S. R. Goldschmidt, J. L. Hennessy, The Accuracy of Trace-Driven Simulations of Multiprocessors, Stanford University, Technical Report CSL-TR-92-546, September 1992

[14] E. J. Koldingr, S. J. Eggers, H. M. Levy, On the Validity of Trace-Driven Simulation for Multiprocessors, in *Conference Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991, Vol. 19, No. 3, pp. 244-253

[15] C. B. Stunkel, B. Janssens, W. K. Fuchs, Address Tracing for Parallel Machines, IEEE Computer, January 1991, Vol. 24, No. 1, pp. 31-38

[16] A. Agarwal, R. L. Sites, M. Horowitz, ATUM: A New Technique for Capturing Address Traces Using Microcode, in *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp.119-127

[17] S. J. Eggers, D. R. Keppel, E. J. Koldinger, H. M. Levy, Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor, in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990, Vol. 18, No. 1, pp. 37-47

[18] C. B. Stunkel, W. K. Fuchs, TRAPEDS: Producing Traces for Multicomputers via execution driven simulation, in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1989, Berkeley, CA, pp. 70-78