The Limes Tool for PC-Based Evaluation of New Architectures

Igor Ikodinovic, Zoran Dimitrijevic, Davor Magdic,
Aleksandar Milenkovic, Jelica Protic, Veljko Milutinovic

This appendix 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 has a scheduling algorithm specially optimized for speed. It can run parallel applications based on the ANL programming paradigm, such as those found in the SPLASH-2 suite. The package also includes an extension for writing object-oriented parallel applications in the Java-like manner. Trace-driven simulation is based on a new concept of ideal traces. The trace-driven simulator 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 the same memory architecture simulators. 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 the Linux operating system.

1. Introduction

Simulation plays a vital role in multiprocessor studies. In a variety of simulation techniques, ranging from analytical modeling, which is often inadequate for being unable to model complex multiprocessor interactions, to hardware prototyping, which is costly and inflexible, software simulation has become relatively 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. In addition, as many simulations as there are available host machines can be run simultaneously, which is important having in mind the number of experiments that usually have to be carried out.

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 the 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 trace-driven simulator and their internal algorithms, and gives insight in one realistic memory simulator on an example of the SMP system. Section 5 deals with Limes complexity and performance. Section 6 gives the brief installation guide for installing and using Limes. Section 7 gives some examples of projects where Limes was used. We conclude with section 8.

[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 execution 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 research that had 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 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 simulators 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 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

Majority of the existing simulation tools is developed for high-end multiprocessor computers. If made for uniprocessor machines, it is almost exclusively for 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 a 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. The only thing that does not allow these tools to be considered perfect, except that they work only on RISC platforms, is the speed of simulation, which 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 (portability was not of primary importance for Limes at the time, but we do consider porting it to other platforms now). 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 the 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. However, our demands were so widely set that the environment we wanted to develop actually represented a general tool for simulations of all shared-memory multiprocessor architectures.

4. Limes Structure

As already indicated, 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. This enables that both execution-driven and trace-driven simulator can use the same memory system simulators. Memory/synchronization references come either from the execution streams of the parallel application's threads or from a trace residing on a disk, depending on the type of the simulation. This is shown in Figure 1. In the current version of Limes (v2.0) only one thread can be assigned to one processor.

The environment also includes an extension of the ANL programming paradigm, so that the users can develop, test, and use as workload object-oriented parallel applications written in C++, but in a manner very similar to that employed by the Java programming language. The whole system is rather small, built on top of Limes. It can be used on any real machine that supports operations defined in ANL macros. The system implements only two classes - threads and monitors, and programs written using them are more readable and understandable than those using raw LOCKs and UNLOCKs. 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.

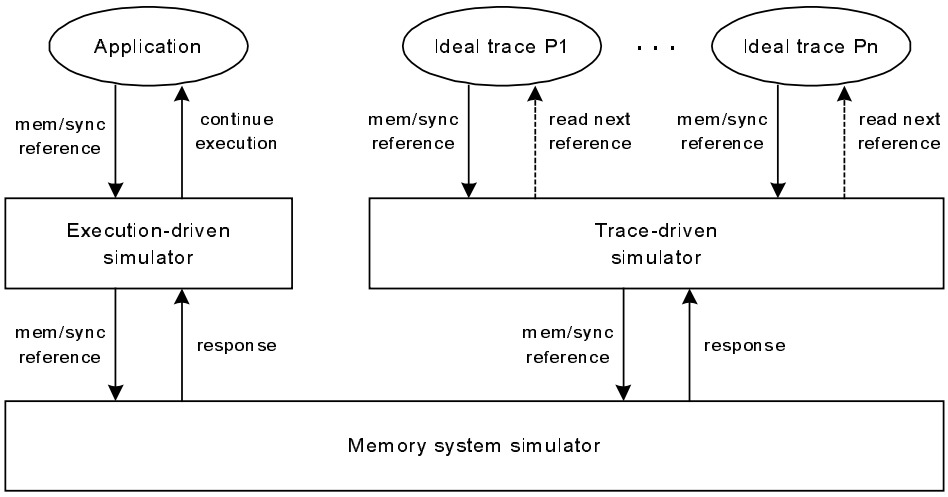

Figure 1. Simulation in the Limes environment

[2] The simulator of the target system actually simulates the behavior 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 system) is not considered here. These are the reasons why this simulator is also called a memory system simulator.

4.1 Execution-Driven Simulator

Execution-driven simulation has two major parts. The first part is *static* - building a simulation executable. The second part is *dynamic* - executing the simulation. Building of the simulation executable will be discussed first. Details of the simulation will be described after that. At the end, it is explained how the optimized algorithm for thread scheduling works.

Building an Executable

Applications that can be used to drive this simulator are like those from the SPLASH-2 suite, or more generally, those that use ANL (Argonne National Lab) macros for expressing parallelism. Application assembly code (created by the compiler) is instrumented with kernel call-outs at compile time; the source code does not require any alterations for that. The simulator allows three levels of assembly code instrumentation, depending on what type of events should be instrumented. 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 simulation speed 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.

Most C/C++ programs call standard library functions, and so do the parallel applications. 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 virtually no impact on the simulation accuracy), but can be, if it is needed; instrumentation tool is open for additions. To avoid collision with library functions that are called by the simulator itself, instrumented functions are prefixed, and a set of macros is defined that enables redirection of calls to the instrumented functions instead of calling the originals. TangoLite, for example, can only instrument those portions of the run-time libraries that are not used by its own run-time system.

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

[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 behavior 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 the 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 discrepancy is less then 1%). CISCs certainly generate more private references, but these references are generally of little importance to multiprocessor studies; shared references is what determines the behavior and performance of multiprocessor systems.

The instrumented application code, simulator, and the memory system simulator are finally compiled and linked together into a single executable. Process of compilation and linking is controlled via a set of make files. If the application code is compiled with an appropriate option, the compiler produces some debugging information that the instrumentation tool can understand. If the application crashes, the simulator will use those pieces of information and print the source line that the offending thread was executing at that moment. It is worthy to know that the instrumentation does not prevent the application to be debugged with a standard debugger.

Simulation

The whole simulation (application threads, simulator kernel, and memory simulator) executes in the context of a single UNIX process. Basically, during the execution the simulator scheduler 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 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 that occurs immediately before an instruction of interest executes is realized through the replacement of such instructions with kernel call-outs at compile time. The programming model supported is the *lightweight threads* model (all processes share the address space with the master process, except for the stack area, which is private for each process). Instruction execution times are calculated at the end of each basic block (a basic block ends with a branch, a label, or a memory reference). They are calculated by adding (a) the time needed for an instruction with a memory reference to complete and (b) the execution times of other instructions in the basic block. 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.

If the simulation is ran with an appropriate option, during the simulation process the simulator will produce a trace that contains references according to the level of instrumentation, along with additional information that can be used to support accurate trace-driven simulation; file format is open for the user to change it.

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, because scheduling activities are very frequent during the simulation. It is important to keep this overhead as low as possible. Other sources of overhead are related to the instrumentation process and to the memory system simulator.

Relative influence of the scheduling overhead on the simulation performance depends on the amount of overhead introduced by the other two factors. Having in mind the frequency of scheduling activities, it is certainly worth-wile to reduce it as much as possible.

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. Therefore, 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. This optimization regards the execution stream of the non-global instructions. The scheduling algorithm forces the continuous execution of non-global instructions as much as possible, saving thus the time needed for frequent invocations of the scheduler. The scheduler will call the memory simulator only if there are no threads that can continue execution, because they all wait for a memory operation to complete. An additional optimization regards shared writes. 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.

The scheduler works as follows: When a memory reference is met in a thread's execution stream, the scheduler is invoked. Scheduler then does not invoke memory system simulator right away, but instead checks if there are any threads that have become free to continue execution, due to the fact that their memory requests have been satisfied in the meanwhile. If there are such threads, then one of them with minimal time (there may be more than one) is scheduled to continue its execution, and the scheduler will wait for some other thread to call it again. If there are no threads that can continue execution, that is, if all of them are waiting for some memory request to complete, then the scheduler enters a loop where he calls the memory simulator in every cycle, until some thread gets its request satisfied and enables it to continue execution. Such a scheduling algorithm imposes that the memory simulator must be called with two parameters, *time_now* and *last_simulation_time*. Memory simulator then performs simulation between these two moments. If *time_now>last_simulation_time* it means that at *last_simulation_time* all threads' requests were satisfied. However, it does not mean that the memory simulator can just skip this time. This is due to the optimization that is performed when simulating shared writes. When a shared write occurs, the memory simulator will return a *satisfy* signal right away, even if the actual write was not simulated. This way a thread can continue execution without delay. The actual shared write will be simulated later, upon some other call to the simulator.

It depends on the memory simulator how many shared writes it can buffer before a coming write must wait for the first one in the buffer to be actually simulated. This is the reason why, upon every call, the memory simulator first checks if there are any unfinished shared writes, and simulates them first if there are some. After the simulator establishes that there are no more unsimulated events, it can freely skip the rest of the time, all up to *time_now*.

4.2 Trace-Driven Simulator

Making a trace of some program's execution enables that a simulation can be split in two phases - local work and a memory system simulation. Local work is not of importance for multiprocessor studies. Trace-driven simulation performs only the memory simulation part, while local work is already built in a trace via time stamps. If multiple simulations are performed with the same trace then trace-driven simulation can potentially bring speedup over multiple execution-driven simulation runs. But due to the disk transfer rate limitations this is not always possible. Multiple runs can 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 with trace-driven simulation. Trace-driven simulation also enables the use of traces that contain OS references, while execution-driven simulations can not include OS references.

There is one more benefit when 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 on one RISC and one CISC platform, would give completely different results for the same workload. Same traces, on the other hand, contain same local work, so the comparison is reduced only to the memory systems that are both equally simulated.

Limes' trace-driven simulator uses as input a trace generated by the execution-driven simulator from the Limes environment, or from some other source. Limes is very flexible concerning the trace file format. Its current version supports two formats: a text trace file format, that can 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 module of the trace-driven simulator (and the module for trace generation of the execution-driven simulator, if traces are produced with it). Every format, however, should be able to support *abstraction* by allowing certain additional information for each reference to be stored in a trace.

Limes enables accurate trace-driven simulation, where possible, by using the method of abstraction to eliminate timing dependencies. The technique used to support abstraction in Limes is original. The simulator currently supports the abstraction of shared reads/writes, locks/unlocks, user defined instructions (like prefetch, forward etc.), barriers, and thread creation events. It is, of course, open for changes and ready to support abstraction of other timing dependent operations, and of different implementations of currently supported primitives.

Accuracy issues

The execution path of a multiprocessor workload depends on the ordering of events in a system, which in turn depends on the machine memory system timings. 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 trace itself usually does not contain any information about program's execution path.

Those pieces of information are implicitly built in a trace via time stamps, and are valid only while the memory system is not changed. If the memory system is changed, timings will change as well, and time stamps will no longer contain correct information about the program's execution path.

In [12] authors find that traditional 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 (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 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 for the first issue, the trace we once obtain can be used for simulation of different memory architectures, so we do not need multiple simulation runs to obtain different traces for different architectures. We can thus achieve accurate simulation, if the trace contains all the necessary information. 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 the same effect, too. 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.

Abstraction

Abstraction of operations that can influence program's execution path means that some information about them is 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, but 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)
- Other timing dependent operations like thread creation

Timing dependent operations other than those that can influence the program execution path (like those involving real-time clock[4]) can also be abstracted.

We divide timing dependent operations in two groups: elementary and complex. They are abstracted in a different way. Elementary operations are those that are realized in the simulated hardware as primitives. Complex operations are realized through the use of elementary operations. In our case, elementary operations are shared reads/writes, lock acquires/releases, and user defined instructions, while barriers are realized as complex operations.

Timing dependencies can be eliminated from elementary timing dependent operations by recording their completion time. That way the simulator knows the amount of pollution introduced by the architecture on which the trace was generated, and eliminates it.

Complex operations are much harder to abstract, depending on how "complex" they are. Barrier implementation via locks, used in Limes, is an example of a complex operation that needs a significant support on the side of the trace-driven simulator. Apart from that, the trace itself must contain rather detailed information on each barrier which has to be provided by the trace generation environment.

In general, it is up to the trace-collecting tool to save all the necessary information in the trace. The abstraction technique we use is responsible for performing accurate simulation based on this information.

Concept of an Ideal Trace

The abstraction of elementary timing dependent operations can be reduced to simply decreasing their time stamp values for the amount of their completion time. In another words, time stamps, that implicitly contain information about the execution path, are reduced to contain *only* those information and *no* information on the memory system. The abstraction can be done this way by postprocessing the trace after it was generated, and before the trace is used for the simulations, which is much more efficient than if the whole procedure was done by the simulator at run-time. This also means that no additional information on those operations is needed in a trace after postprocessing, which reduces its size. 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 (so no additional data are needed by the simulator), and where complex timing dependent operations have been properly abstracted.

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

The easiest way to produce it is to run a simulation using an ideal memory simulator. With ideal memory simulator every memory request can be completed in a single cycle. However, lock acquire operations will not be abstracted that way (for they must preserve the correct global ordering). We fight this inconvenience by forcing every attempt for gaining a lock to appear in a trace until it is finally gained. 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.

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. But although this format is good for keeping traces, it is not good considering the simulation efficiency. 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 can be done only once, and the obtained ideal traces can be then used for multiple simulations.

The simulation starts by reading the first reference of each processor's thread and by scheduling the one with the smallest time stamp, and continues with scheduling requests until the last one is simulated. The process is quite straightforward, except for the barriers. In the current implementation each barrier in a trace is just a set of reads, writes, locks and unlocks. There would be no way to tell if they functionally execute this synchronization primitive, 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 using those 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 those 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 memory system simulator is completely independent from the simulation kernel and from the applications (traces) used to drive the simulations. As mentioned before, it can be used by both the execution-driven and the trace-driven simulator.

The simulator provides the memory system simulator with a stream of requests. They use a simple interface to communicate, and the same interface should be used when building up a simulator for any kind of shared-address space multiprocessor system. How is the memory simulator realized is of no importance as long as it uses that interface.

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 an on-chip (L1) cache implementing one of the five supported coherency protocols. The processors are interconnected via bus, and all the communication through bus signals is adequately mimicked. The main memory module is also connected to the bus. This organization is represented in Figure 2.

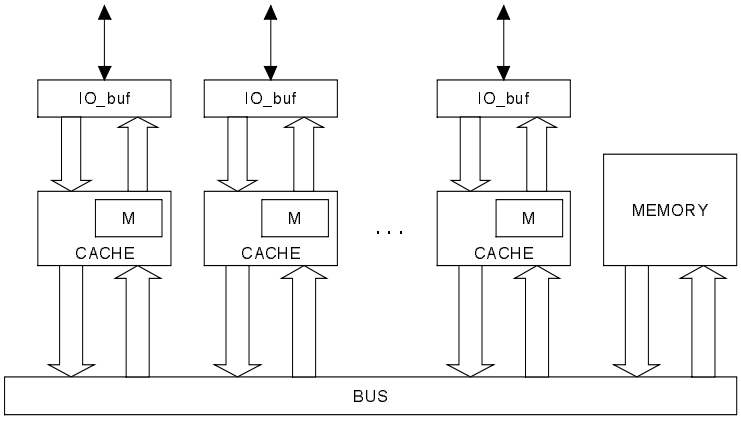

Figure 2. Interconnection of modules in a bus-based cache coherent symmetrical multiprocessor system

Legend: CACHE - cache controller module; M - cache memory module; IO_buf - simple abstract modules that communicate with simulation kernel; MEMORY - main memory module; BUS - the bus.

Each hardware unit is programmatically represented with a module (a C++ class), and each module is unaware of the others. 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. These ports are shown in Figure 3.

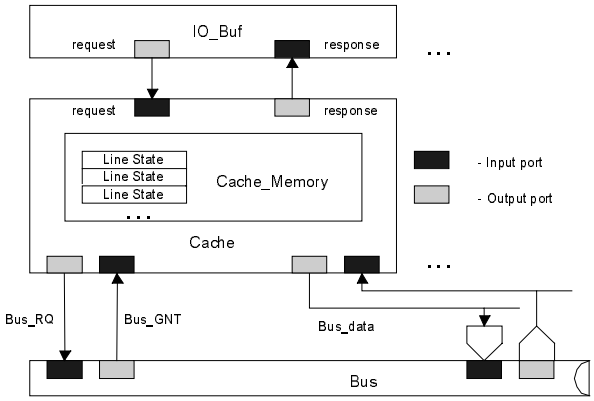

Figure 3. Cache module and input/output ports

Only one (abstract) module in the system is aware of the system topology, and it is not visible in the picture. This module reads output ports from the modules and writes the information found there into the input ports of the modules they are connected to, following certain order and simulating signal flow; it also polls the modules for execution at proper times. This organization would allow, for example, to easily add the TLB modules between the cache controllers and the processors without changing other modules (except the topology module, of course).

This design philosophy resembles greatly the one employed by VHDL. It allowed us to retain the desired level of accuracy, while preserving the simulation speed and the ability to change the model easily. 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.

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. Cache controller module is represented as a finite state machine. Since all modules operate on a cycle-by-cycle basis, they change their internal state according to the current state and the information on their input ports. However, these states do not have to be understood in a strict sense as defined by the finite automata theory. The operation of the module is represented as a mixture of a flow chart and a state diagram. Modules are easily modifiable and extendible since they are written in C++, employing the OOP style.

DSM Memory Simulator

A simple model of a distributed shared memory multiprocessor system (DSM) is developed for Limes. The system comprised of N identical processors. Each processor has local memory, which is a part of the global distributed shared memory. Several memory consistency models are implemented in order to simulate the access to DSM. The system currently includes examples of processor, sequential, release, and lazy release consistency. The underlying network delays can be simulated in details.

In order to simulate a DSM multiprocessor system, a specific memory simulator is developed. Different memory allocation strategy is needed for DSM systems, and some parts of the old Limes kernel had to be changed. The user can easily change the strategy for memory allocation (G_MALLOC macro). The memory simulator consists of the following parts: distributed shared memory itself, directory simulator, lock simulator, and network latency simulator.

Size of the distributed shared memory is statically defined and distributed between the nodes. Current implementation of memory allocation enables user to specify home node for memory block to be allocated by the G_MALLOC macro. The global and static variables are by default allocated in the DSM space local to the first node. If the home node is not explicitly specified in the call to the G_MALLOC macro, then it is assumed that the home node for the new block is the node which called the macro.

The directory simulator implements the memory consistency model for distributed shared memory access. This simulator is called by the kernel every time the read or write instruction occurs. The simulator is responsible for all local page or block caching and acquiring of non-local DSM data. For the implementation of new consistency models this part needs to be changed.

In DSM systems lock management is quite different than the one in bus based systems, because there is no bus to make the serialization. There are several different approaches to implement locks in DSM systems. However, all implementations make data access to the memory storing the locks sequentially (or at least processor) consistent. This simulator is called by the kernel every time the lock related operations occurs (acquire or release).

Network latency simulator can be used to simulate different types of interconnection networks in details. However, this part can be as simple as making the different constant delays for access to non-local parts of the distributed shared memory. If the aim of the research is to simulate the specific network configuration, this is the only part of the Limes that the researcher has to change.

Virtual Address Space Layout in SMP Limes

When multiple threads of execution are created (for simulation purposes by Limes, in reality by any of the newer Linux releases), they are defined as regular processes that share address space with their parent. More formally, page table entries for mapping virtual addresses to physical have the same values for all the threads. Figure 4 depicts this situation for an example of four threads.

All the threads share the code segment (of course) and the data segment, and each one gets its reasonably wide share (say, 256K) of the upper area for the stack. The tops of their stacks are different. Stacks are private only semantically: a thread *could* read and write another threads' stack, but never does so.

This brings us to the point: why do you not have to translate virtual addresses to physical in simulation? The answer is – because no two different virtual addresses point to the same physical address, and no two equal virtual addresses point to different physical addresses.

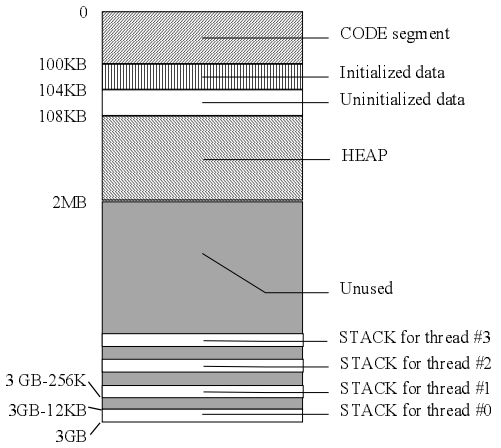

Figure 4. Virtual address space layout for multiple threads in SMP systems (an example).

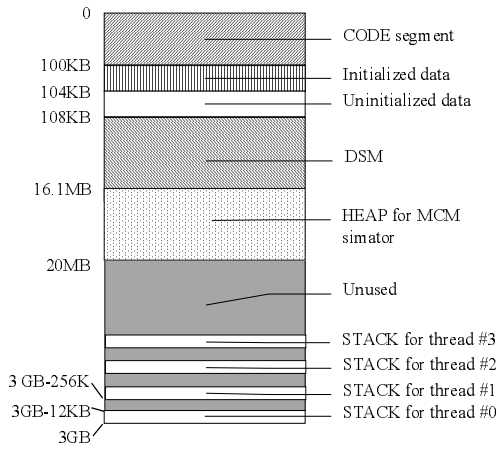

Figure 5. Virtual address space layout for the simulation of DSM systems (an example).

Because of that one can assume that these threads of the parallel application are the only set of processes that are executing on your simulated machine. And one can also assume that the code segment was really loaded into memory at address zero, and so on.

There are certain leaks in this assumption: first, one does not have 3GB of physical memory to assume that the stack is physically really placed at 3GB minus something; and second, in reality, the lowest part of the physical memory would be assigned to system tables, etc. But since all the allocations occur at page boundaries, it is only a matter of distribution of block tags in your associative memory in caches, which is unpredictable anyway, and has little impact on general behavior for 2-way caches or caches with higher associativity, or larger caches. Still, it would be easy to create a translation function, should one needs one.

Finally, a note on the **ISSHARED(addr)** macro: It returns true if (and only if) the address given belongs to the **data** segment. Formally, code segment is shared as well; but it can be read only, and is therefore not important from the aspect of coherence maintenance. Note that reads from the code segment can occur, although Limes does not capture instruction fetches; this is because the application may read constants embedded in the code segment, mainly string constants, for various `printf()` functions that never occur in the parallel computation phase. If you really need to know whether an address belongs to the code segment, say `if(addr < DATA_SEGMENT_START)`.

Virtual Address Space Layout in DSM Limes

The consistency models in DSM systems are more relaxed than in SMP systems, and the DSM memory is less consistent than the SMP memory. In order to accurately simulate the system, fetching of the non-local data is simulated in details. A node in a DSM system holds a part of the distributed shared memory, and a part of the local, non-shared memory used for storing and caching of the non-local data and the directory data used to maintain the DSM consistency.

The virtual address space of the DSM simulator is shown in Figure 5. Directory simulator dynamically allocates and frees the memory in the heap for the simulation of the MCM during the simulation.

The main difference between the SMP and the DSM simulation virtual address space is that the space for the DSM is statically allocated at the beginning of the simulation and the G_MALLOC and G_FREE macros are re-implemented in order to correctly allocate and free the space in the DSM. This enables the user to implement new algorithms for memory allocation in DSM systems.

5. Limes Complexity and Performance

The simulation kernel of the execution-driven simulator consists of nearly 1800 lines, and the tool for instrumentation of iAPx86 instructions contains additional 1200 lines. Trace-driven simulator adds 1500 lines. The SMP model and five snoopy protocols comprise alltogether some 4000 lines of code. The most complex classes – the detailed cache controllers, contain on the average some 400 lines. The whole code is profusely commented. The complete Limes environment (including 9 SPLASH-2 applications) takes about 820KB (compressed).

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

Limes performance for execution-driven simulations is presented in Table 1. The results show execution times and slowdowns for four SPLASH-2 programs. 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. All simulations were performed for 16 processors, where each had a 64KB large, 2 way set-associative cache, with 16B long cache lines.

simulator / level	OCEAN		FFT		LU		RADIX		Avg.
	time [s]	slowd	time [s]	slowd	time [s]	slowd	time [s]	slowd	slowd
uninstrumented	8	1	2	1	16	1	7	1	1
abstract/level_1	312	39	65	32	863	54	77	11	34
abstract/level_2	684	85	94	47	1120	70	270	38	60
ideal/level_1	729	91	122	61	2438	152	132	19	81
ideal/level_2	1334	166	200	100	3045	190	698	100	139
MESI/level_1	5273	660	850	425	10056	628	737	105	454
MESI/level_2	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 simulator models

Dimensions of the problems were appropriate: OCEAN worked on a 130x130 grid, FFT with 65536 complex doubles, LU with a 512x512 matrix and RADIX with 262144 integers. Simulations were done on a 133MHz Pentium PC.

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, introducing an average slowdown factor of 34 for level_1 and 60 for level_2 instrumentation. Slowdowns using ideal memory simulator indicate the scheduling plus the memory simulator invocation overhead. It additionally slows down the simulation for a factor of 1.7-2.8 (2.4 in average) for level_1, and for a factor of 2-2.7 (2.3 in average) for level_2 instrumentation. Finally, simulation of a realistic bus-based SMP system with a MESI protocol indicates the memory simulator overhead. For level_1 simulations it brings a slowdown that goes from 9.5 to 16.9 (13.3 in average) and for level_2 simulations the slowdown ranges from 6.7 to 9.5 (8.6 in average) in addition to the instrumentation overhead. Pure memory simulator slowdowns, not counting the scheduling and the simulator invocation overhead, go from 4.1 to 7.9 (5.6 in average) for level_1, and from 2.6 to 4.5 (3.7 in average) for level_2 simulations.

simulator / level	OCEAN		FFT		LU		RADIX		Avg. overh %
	size [KB]	overh %	size [KB]	overh %	size [KB]	overh %	size [KB]	overh %	
uninstrumented	167.8	0	121.5	0	125.1	0	120.7	0	0
abstract/level_1	237.9	41.7	138.3	11.4	137.8	10.1	129.3	7.1	17.6
abstract/level_2	323.9	93.0	154.7	27.3	158.3	26.5	141.6	17.3	41.0
ideal/level_1	237.9	41.7	138.3	11.4	137.8	10.1	133.4	10.5	18.4
ideal/level_2	323.9	93.0	154.7	27.3	158.3	26.5	145.7	20.7	41.9
MESI/level_1	357.5	113.0	257.9	121.2	257.4	105.7	248.9	106.2	111.5
MESI/level_2	443.5	164.0	274.3	125.7	277.4	121.7	261.2	116.4	132.0

Table 2. Sizes of execution-driven simulation executables and memory dilation overheads for 4 SPLASH-2 applications, for various memory simulator models.

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

Table 3. Speed of the trace-driven simulation for 4 SPLASH-2 applications compared to the execution-driven simulation. The table shows simulation times for both simulators (running the same workloads) and their ratio.

These results are comparable to the TangoLite performance. TangoLite has an average slowdown of 45 for a simulation with no memory simulator, which compares to an average Limes slowdown of 34 in the abstract/level_1 case. An average memory simulator slowdown for Limes is 13.3 for MESI/level_1, and for a similar memory architecture TangoLite has a slowdown factor of 17. In total, Limes has a slowdown of 454 compared to the TangoLite average slowdown of 765.

Instrumentation inevitably increases the size of the application. Table 2 shows some values of memory overhead that Limes introduces. These results can be compared to the TangoLite overhead, where instrumentation typically increases application static size by a factor of four, against the factor of 2.1-2.3 for Limes.

Results for trace-driven simulation are presented in Table 3. They show the trace-driven simulator performance against the execution-driven simulator. Simulation times are obtained using small, but sufficient examples. OCEAN, FFT, LU, and RADIX binary traces were 8.7MB, 9.0MB, 18.3MB, and 11.7MB long, respectively. Disk transfer rate is about 1.0 MB/s. Simulations were performed for 2KB large on-chip cache, for 8 processors.

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.

6. Installation Guide

One does not need any privileges to install and run Limes, modulo the installation of version 2.6.3 of the GCC compiler, if it is not installed on the system already. The procedure is described below. For the purpose of simplicity, it will be assumed that the username is **joe**, and that the home directory is **/home/joe**.

6.1. Unpacking the Archive

Limes comes in a single archive, **limes*.tgz**, where * stands for the current version. While this procedure describes how to install Limes in a per-user manner, you can also choose to install it on the system to be globally accessible. To unpack the archive, simply position in your home directory and do

```
tar xfz limes*.tgz
```

After this, a directory tree will be created.

6.2. Setting up the Environment

Limes scripts and makefiles must know where the whole tree is. To communicate this information, set the environment variable **LIMESDIR** to point to the root of the tree. In the example, do

```
export LIMESDIR=/home/joe/limes        # if you use bash, or
setenv LIMESDIR /home/joe/limes        # if you use tcsh
```

(You can put them in your .profile or .login for convenience).

6.3. Installing the GNU C v2.6.3 Compiler

Check the version of your GCC compiler with "gcc -v". If it reports **2.6.3** or lower, the installation procedure is over. Otherwise, you will probably have to be root (or to ask one) to copy the development environment for the GCC 2.6.3 version on your filesystem. **Note: do not really *install* it, for it will overwrite your current GCC! Just read below how to *copy* it onto your filesystem, in a separate, harmless directory! Limes will know how to find and execute it.**

As explained, Limes does various things with the application's assembly code, and is therefore strongly dependent on the version of the compiler. But the 2.6.3 version does not have to be the default compiler on your system, of course. It will reside in a sub-directory and be called only by the Limes. Here is how to do it:

1) First, get the Development disk series of the Linux distribution that contains GCC 2.6.3. An example is Slackware 2.2.0.1
2) You will need the following archives: **gcc263.tgz**, **include.tgz**, **libc.tgz**, **libgxx.tgz**, **lx128_2.tgz**, **binutils.tgz**.
3) Create a directory, say, **/oldgcc**. Change the ownership of this directory to make it belong to a non-privileged user. Now login as that user and cd to /oldgcc.
4) From now on, everything will be done from the directory /oldgcc as the current, and all the paths will be relative to that directory.
5) Unpack the archive d1/gcc263.tgz (like `tar xfz /cdrom/slackware/d1/gcc263.tgz`)
6) Unpack d6/lx128_2.tgz. Do
```
(cd usr/src/linux-1.2.8/include; ln -sf asm-i386 asm)
(cd usr/src; ln -sf linux-1.2.8 linux)
```
7) Unpack d2/include.tgz. Do
```
(cd usr/include;
ln -sf /oldgcc/usr/src/linux/include/linux linux)
(cd usr/include;
ln -sf /oldgcc/usr/src/linux/include/asm asm)
```
8) Unpack d5/libc.tgz. Do `sh install/doinst.sh`
9) Unpack d4/libgxx.tgz. Do `sh install/doinst.sh`
10) Unpack d8/binutils.tgz

There is one more step: edit **limes/globals.make** file in your Limes tree, and replace "ifeq (0,1)" with "ifeq (1,1)". If you have chosen another directory for GCC2.6.3 instead of "/oldgcc", change that in the file, too.

The procedure is now over. It will not affect the rest of your system, except that you will now have 11MB of disk space less.
If you do not have access to a Linux distribution that contains GCC v.2.6.3 files, you can download it from the following URL:

`http://galeb.etf.bg.ac.yu/~dsm/limes/oldgcc.html`

(or you will find a pointer to a site that contains it). It is in fact somewhat reduced, and requires only 7MB of space (the archive itself is 2.4MB long).

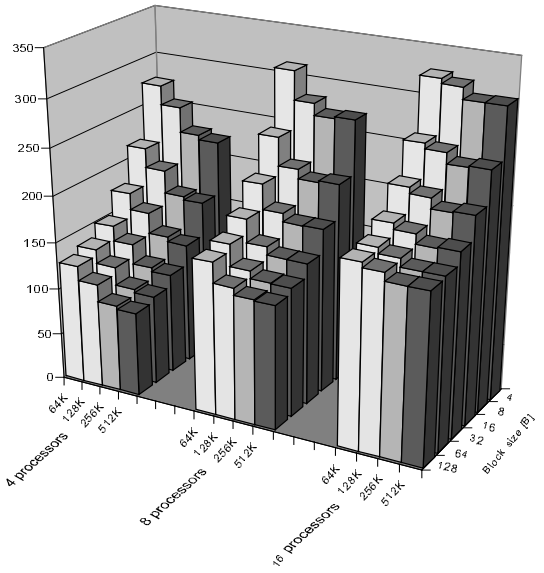

Figure 6. Measuring bus traffic with Limes. This figure shows values (in MB/s) for FFT application.

7. Experiences with Limes

Limes has been used for a number of studies in the domain of shared-memory multiprocessors, including SMP and DSM [19] systems.

In one of the studies regarding characterization of parallel applications for DSM systems [20], Limes was used as a simulation environment, and one originally developed characterization tool was used to obtain parameters of interest. The data obtained by Limes were then used in two other studies - as an input to the analytical models of DSM systems, and to assist researchers in enhancing performance of DSM systems by improving memory consistency protocols.

In another study regarding SMP systems Limes was used to evaluate influence of different architectural parameters on the overall system performance. In particular, bus traffic and miss rate were measured on a system with a MESI cache coherence protocol while varying the number of processors, cache size and cache block size, and using different workloads. The results were then used to infer some conclusions regarding the architectural details. As an example, values for bus traffic are shown in Figure 6 (for FFT application from the SPLASH-2 suite, supposing 100 MHz processors). Bus traffic decreases as cache size grows, because there are fewer cache misses that induce bus traffic. Bus traffic also decreases with the growth of cache block size, because FFT exhibits spatial locality, so there is no negative influence of false sharing. When the number of processors grows, bus traffic increases, as there are more requests on the bus.

In the recent study concerning implementation of the lazy release consistency model in SMP systems, Limes was successfully used to simulate the environment that supports the proposed improvement in hardware, and to show some quantitative results.

8. Conclusions

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 those who need realistic simulations of shared-address space multiprocessors (for architecture evaluation, in investigating real-time systems [21], etc.), and to the researchers in the field of parallel algorithms.

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 a relatively fast type of simulation that still preserves correct global ordering; also, a new paradigm for easy and comprehensible parallel programming is available.

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, and trace compaction techniques [22] may be used. 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. Simulators of some other shared memory systems are currently being developed and will be included in the environment. In the end, Limes should soon be ported to work on other platforms and operating systems.

Acknowledgements

The authors greatly appreciate the help from Professors Milo Tomasevic and Igor Tartalja of the University of Belgrade, and Darko Marinov of MIT. Of course, credits also go to the users of the package on universities around the world whose feedback aided the authors in improving the tool: K. C. Liu and Hung-Chang Hsiao of the National Tsing Hua University in Taiwan, Yoon-Jung Hur of the Korea University in South Korea, Jawad Khan of the GIK Institute of Technology in Pakistan, Artur Caetano of the University of Lisbon in Portugal, Valentin Puente of the University of Cantabria in Spain, Jose Flich Cardo of the University of Valencia in Spain, Brian Boysen of the Clemson University in USA, Brian Grayson of the University of Texas at Austin in USA, Jeff Heid and Denis Reilly of the Carnegie Mellon University in USA, and Fabrizio Petrini of the University of California at Berkeley in USA. Initial ideas and implementations of Limes came from Davor Magdic.

Availability

The whole Limes package is in the public domain, and can be found at the following Internet address:

http://galeb.etf.bg.ac.yu/~dsm/limes

References

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

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

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

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

[5] Milutinovic, V., *Issues in Microprocessor and Multimicroprocessor Systems: Lessons Learned*, Wiley, New York, New York, USA, 2000 (http://galeb.etf.bg.ac.yu/~vm/books/2000/mm.html).

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] Protic, J., Tomasevic, M., Milutinovic, V., "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel and Distributed Technology*, Vol. 5, No. 1, Summer 1996.

[20] Marinov, D., Magdic, D., Milenkovic, A., Protic, J., Tartalja, I., Milutinovic, V., "An Analysis of SPLASH-2 from the DSM Point of View," *Proceedings of the Hawaii International Conference on System Sciences*, Mauna Lani, Hawaii, USA, January 1998.

[21] Stankovic, J. A., Ramamritham, K., "Advances in Real-Time Systems," *IEEE Computer Society*, September 1993.

[22] Samples, A. D., *Mache: No-Loss Trace Compaction*, Performance Evaluation Review, Vol. 17, No. 1, pp. 89-97, May 1989.

[23] Milutinovic, V., "The Best Method for Presentation of Research Results," *IEEE TCCA NEWSLETTER*, pp. 1-6, September 1996.