Trojan: A High-Performance Simulator for Shared Memory Architectures *

Daeyeon Park and Rafael H. Saavedra
Computer Science Department
University of Southern California
Los Angeles, CA, 90089-0781
{daeyeon, saavedra}@cs.usc.edu

Abstract

This paper presents an execution-driven simulator called Trojan, which is an extended version of MIT Proteus, for evaluating the performance of parallel shared-memory machines. The key features of Trojan are: 1) it simulates efficiently both process-model based (e.g., SPLASH) and thread-model based applications (e.g., SPLASH2) (a “copy-on-write” mechanism is used on process-based applications implemented on a threads package); 2) it provides support for virtual memory simulation, which is, to our knowledge, the first execution-driven simulator to offer this functionality; and 3) Trojan does not require making any modification to applications, which results in increased accuracy and usability. We have used Trojan extensively to study cache behavior, network traffic patterns, multiprocessor architectures, and application behavior.

1. Introduction

Although execution-driven multiprocessor simulators have been shown to be effective in simulating large programs running on complex machines, sometimes these simulators tend to offer a limited set of features which can potentially restrict their applicability. The three most important features to consider are: 1) the specific shared-memory execution model supported; 2) the ability to simulate virtual memory; and 3) whether or not applications need to be modified in order to conform to the simulator requirements.

With respect to the execution model, simulators tend to support one of two models: process-based (e.g., SPLASH [15]) or thread-based (e.g., SPLASH2 [16]). The difference between them is that, in the process model, heavyweight processes created with the Unix “fork” receive their own private copies of all global variables (private global and not shared global variables), whereas in the thread model, lightweight processes share the same virtual address space, and hence all global data. Most simulators use the same model as the applications they simulate (i.e., Unix processes under the process-model) and thus support only a single set of semantics. This, however, presents two problems: first, a separate simulator is needed for each model, which is inconvenient and expensive, and second, simulating a process model using Unix processes is very expensive mainly due to the high cost of context switching. For example, the Stanford Tango simulator [17] supports only the process model and can only simulate applications sharing these semantics. In contrast, Stanford’s Tango-Lite [18] (a successor to Tango) and MIT’s Proteus [14] use threads to simulate only thread-based applications. In order to simulate process-based applications in these simulators, all private global variables (variables visible by a single process/thread) which can be modified during the parallel stage have to be replicated (manually modified) [18]. This imposes a significant burden on the user.

Another important limitation of many simulators is their lack of support of virtual memory execution. While most Massively Parallel Processing research machines (MPPs) do not support virtual memory, recent commercial MPPs such as KSR and Intel Paragon do [8, 13]. Furthermore, as networks of workstations (NOWs) emerge as an alternative to dedicated MPPs, the performance impact of virtual memory features such as TLB misses, page faults, and working sets has become more significant. Thus, we believe that in order to reflect more realistic behavior, future simulators will need to offer some support for it. This, however, has been ignored in most state-of-the-art simulators.

A third important limitation of many simulators is that they sometimes require that applications be annotated in order to simulate them correctly. For example, Proteus uses an extra symbol to represent accesses to shared memory. Though this simplifies writing the simulator, it imposes

---

*This material presented here is based on research supported in part the National Science Foundation under grant CCR-9308981, by the Advanced Research Projects Agency under Rome Laboratories Contract F30602-91-C-0146, and by the USC Zumberge Fund. Computing Resources were provided in part by NSF infrastructure grant CDA-9216321.
cache, a portion of the shared memory, a directory, and a network interface, all of them linked by a bus. Processors are connected directly to their respective caches. The physical memory is equally distributed among the nodes and a particular memory consistency model (sequential, relaxed, etc) [5] is enforced by the hardware. Cache coherence for shared memory is maintained using a distributed directory-based protocol. We believe that this is a reasonable abstraction for many large scalable shared multiprocessors (e.g., Stanford DASH [4], Stanford FLASH [7], and MIT Alewife [2]), since it captures the essence of the memory hierarchy of such machines. Trojan applications are written in C using the ANL macros [3]. These macros provide a variety of abstractions such as locks, barriers, and process (or thread) management.

2.2. Execution-Driven Simulation

There are the two well-known methods for parallel computer simulation: trace-driven and execution-driven simulation. Trace-driven simulation decomposes the system into two components: a trace generator and a memory/network simulator. The simulator emulates the execution of the target machine based on the addresses the trace generator produces. Though trace-driven simulators are relatively easy to build, their accuracy is limited by the timing assumption the address generator makes [17]. Execution-driven simulation, on the other hand, interleaves the execution of the application program with the simulation of the target system, without having to generate an intermediate address trace. This interleaving (unlike timing assumptions in trace-driven simulation) allows more accurate simulation of contention and interactions amongst the nodes [17].

Execution-driven simulation with a threads package allows one process to be multiplexed amongst the various activities in the target machine. Sequences of instructions are directly executed until the program performs globally visible operations such as shared memory references. When this happens, the simulated time is updated and control is returned to the simulator engine for the scheduling of future events. The simulator is the main thread; it maintains a queue of requests, sorted by timestamp. While the queue
Tango supports trace-driven simulation as well as execution-driven simulation. In an execution-driven mode, the Tango simulator uses the Unix “fork” to create child processes. Though this scheme makes simulating process-based applications relatively easy, the overhead of context switching tends to dominate the simulation time because executing one tends to consume thousands of cycles. Frequent context switching between processes is necessary in order to accurately interleave the execution of events. To reduce this overhead, Tango allows the user to select the regions of memory to simulate in detail: all memory, only shared memory, or only synchronization operations.

Whereas Tango runs in multi address spaces, Proteus, on the other hand, runs in a single address space, making it two orders of magnitude faster than Tango. Moreover, Proteus can be easily configured to simulate a wide range of architectures such as a bus-based or k-ary n-cube networks. Proteus also provides detailed hop-by-hop network simulation as well as analytical modeling using Agarwal’s network model [1]. Two limitations of Proteus are that process-based applications cannot be simulated and that applications have to be changed to represent shared memory references.

Compared with Tango and Proteus, Trojan has some advantages: first, it supports efficiently the two basic execution models with a threads package. Second, Trojan provides functionality for virtual memory simulation whereas Tango and Proteus do not. A disadvantage of Trojan relative to Tango is that Trojan does not support trace-driven simulation. Compared with Proteus, Trojan has the advantage of not requiring any modification to applications.

4. Simulation of Two Shared Memory Models

The semantics of parallel shared-memory programs are represented by two different execution models and three different scoping levels. This section explains in detail the
process model and global in the thread model. Thus, in order to accurately simulate parallel programs in a uniprocessor environment, we need an extra scope as a way of representing global variables in the process model and private global variables in the thread model. Global variables (i.e., shared memory) in the process model are represented by `global malloc()` instead of `malloc()`, and allocated only from the reserved part of the address space. For private global variables in the thread model, there is no easy way to express them and thus several awkward methods have been used in the past. One way is to allocate a chunk of memory for these variables and pass it to each function in the thread, as an extra parameter [12]. Thus, although the thread model has an advantage of making communication simple and efficient, it has the disadvantage that it tends to overly complicate writing the simulator.

4.2. Simulation of the Two Models

Most simulators are developed with a specific execution model in mind and thus support only a single programming model. This approach, however, has two problems: first, when we need to evaluate the performance of benchmarks written in the two models (as each model has its own advantages and disadvantages) we cannot use a single simulator for both models. The fact that a simulator is needed for each model causes many inconveniences and is expensive. The second problem is that of the high overheads we have to pay when we simulate processes/threads using Unix processes. When a child process is created, the whole memory image of the parent process is copied into the child address space. A naive implementation of this copying scheme requires huge amount of extra memory and consumes simulation time. To this we have to add the high overhead of context switching which is incurred in interleaving the execution of the simulated parallel processes. Context switching requires thou-
Table 1. Examples of Execution-Driven Simulators and Programming Model

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Supporting applications</th>
<th>Simulation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tango</td>
<td>Process model</td>
<td>Unix processes</td>
<td>high ctx switch overhead</td>
</tr>
<tr>
<td>Proteus, FAST</td>
<td>Thread model</td>
<td>Threads package</td>
<td>No support for process model</td>
</tr>
<tr>
<td>Trojan</td>
<td>Process &amp; thread model</td>
<td>Threads package</td>
<td>Copy-on-write</td>
</tr>
</tbody>
</table>

sands of cycles, making this overhead the dominant factor of simulation time. Table 1 summarizes the programming model that some of execution-driven simulators support.

To deal with this problem, Trojan uses a threads package to simulate programs representing both the process and the thread model. The fact that all target processes run in a single address space solves the problem of heavy context switching, but creates a new problem: how to enforce the semantics of the process model. That is, how to make updating private global variables visible only to a single process. One solution is that each thread (simulating a target process) makes a whole copy of the parent process. However, this scheme requires huge amount of memory and copying time.

To get around this problem, Trojan uses a copy-on-write scheme [12]. Under a copy-on-write policy a page is copied only when the child process tries to write into it for the first time. As long as the child makes only read references to the page, the operation is satisfied using the original page. However, if the child attempts to write into the page, the simulator creates an independent copy. Subsequent references to this page are done on the copied page.

Figure 4 illustrates how Trojan’s thread package simulates a process model using copy-on-write. In Fig. 4, child A makes read references to page 0, 1, and 2, while child B makes read references to page 0, no reference to page 1, and read-write references to page 2. In this example, no independent copies of the pages exist for child A, but when child B attempts to write into page 2, Trojan copies this page into a new page (represented by page 9). Copy-on-write has several advantages over copying the whole process. First, some pages are read-only, so there is no need to copy them. Second, some pages may never be referenced, so they do not have to be copied. In this way, only those pages that the child process actually writes on have to be copied, resulting in less overhead.

5. Virtual Memory Simulation

Given that recent commercial MPPs such as KSR and Intel Paragon use virtual memory and NOWs (networks of workstations) are now becoming a credible alternative to MPPs, assessing the impact of virtual memory as represented by TLB misses and page faults in the overall performance is critical. Thus, in order to account of these important effects, simulators need to offer some support for virtual memory simulation. For example, without virtual memory, cache simulation is done using the same address generated by the compiler. In many real machines, cache addressing is done using the physical address (unless the cache is virtual). Hence to improve overall simulation accuracy, virtual memory simulation is needed. In this section, we discuss the assumptions we make about the simulated environment in order to support virtual memory and then show how Trojan is able to do this efficiently.

5.1. Virtual Memory Environment

Virtual memory can be implemented in several ways: paging, segmentation, and segmentation with paging. For generality and simplicity, in Trojan we use paging as implemented by the Mach operating system [12].

As Fig. 5.(a) shows, we assume that part of the address space is reserved for shared memory and only these areas are used for this purpose even when, in the thread model, the whole address space can be shared. The main reason for doing this is to simplify the allocation of the cache coherency directory space. Trojan simulates a NUMA machine, in which physical shared memory is distributed amongst the nodes and a distributed directory is used to maintain cache coherency. If the whole address space were allocated for shared memory, directory handling could be much more complicated and expensive. This is because it is very difficult to know whether the memory reference represents shared memory or private memory. Thus we would have to allocate a large directory in order to cover all possible memory references. Hence huge amount of memory would be needed for the directory itself even though no directory is needed for private variables.

5.2. Virtual Memory Simulation

With virtual memory, the CPU produces virtual addresses that are translated by the TLB into physical addresses to access main memory. The main difficulty in simulating virtual memory is how to deal with the physical addresses and the corresponding need for huge amounts of memory. Shared memory simulation without virtual memory does
not cause any special problems since the address comes from the reserved part of a contiguous area for which there exists a corresponding directory. When simulating virtual memory, however, the address given to the memory module is a physical address, and since the physical address can in principle be any address, the directory has to be large enough to cover the whole physical space.

One solution is to limit the mapping of virtual to physical to the contiguous specific address range. This method, however, does not reflect real virtual-to-physical mappings. In addition, the directory handling is still expensive since there is no one-to-one correspondence between pages and frames. That is, a directory is allocated for a frame, not for a page, and a frame can be used for several different pages. Trojan deals with this problem by using different addressing modes at different levels of the memory hierarchy, as shown in Fig. 5.(b). First, Trojan addresses the TLB, page tables and the cache in the same way as it is done in real machines. In the memory module, however, the virtual address is used to access a memory location and its corresponding directory. Although this scheme simulates virtual memory in a partial way, it achieves the same effect as the real virtual memory execution, without requiring additional memory. This is due to the fact that all the performance data for virtual memory can be obtained from the behavior of the TLB, page tables and the cache.

Most of large memory requirement for virtual memory has to do with the amount of space needed for the page tables, e.g., a 32-bit address space with a 4K page size needs 1 million page table entries for each process. This memory requirement is a significant limitation in the simulation of virtual memory.

Trojan addresses this problem by splitting a page table into two parts: one for shared memory and another for private memory. For shared memory, a page table is statically allocated and shared among the processes/threads. Since part of the address space is allocated for shared memory, this page table can be easily calculated and allocated. For 16M of shared memory with 4K page size, only 4,096 page table entries are needed. For private memory, a page table is allocated as needed using a simple open hash function. With $B$ buckets, we use a hash function $h$ such that for each page $p$, $h(p)$ is the bucket to which $p$ belongs. The elements on $i$th list are the page table entries that belong to bucket $i$ and are allocated dynamically at simulation time. Using this scheme, the additional memory requirement for the page table was found to account for less than 5 percent of the total memory requirement.

Currently, Trojan assigns shared pages to nodes by round-robin and private pages reside on its corresponding node. On a page fault, Trojan selects a victim page using the Clock algorithm [12]. This round-robin page allocation and Clock page replacement algorithm can be easily replaced with other policies since Trojan is designed with a modular structure. The virtual memory simulation has been used for the simulation of networks of workstations (NOW) working as a parallel machine.

### 6 Additional Functionality in Trojan

So far we have discussed the issues of simulating two different execution models of shared memory programs using a threads package and the simulation of virtual memory. Another issue relevant here is the requirement that applications be annotated in order to simulate them correctly. Ideally when simulating a parallel program, the source code should be compiled and optimized in its original form as it would be done on a shared memory multiprocessor. However, some simulators like Proteus require making changes to the source code. This changes, however, can be burdensome on a complex application and can cause accuracy problems.

This section discusses the issues involved in annotating ap-
Original assembly code

- add %l3, %l4, %o1
- ld [%i2+ %lo(_A)], %o3

After Augmentation

- add %l3, %l4, %o1
- augmentation of ‘ld [%i2+ %lo(_A)], %o3’
- add %i2, %lo(_A), %l0
- sethi %hi(_mem_address_), %l1
- st %l0, [%l1 + %lo(_mem_address_)]
- mov 2, %l0
- sethi %hi(_mem_type_), %l1
- st %l0, [%l1 + %lo(_mem_type_)]
- call _mem_issue, 0
- nop
- sethi %hi(_mem_return_addr_), %l0
- ld [%l0 + %lo(_mem_return_addr_)], %l0
- ld [%l0], %o3

Figure 6. An Example of Code Augmentation

Applications and in the simulation of private memory. We also discuss other useful features Trojan provides to the user.

6.1. Application Annotations

Proteus is a good example of a simulator that requires adding annotations to the source code in order to simulate it. It requires the use of special operators when accessing shared memory variables. These new operators are replaced by the preprocessor with specific simulation calls. This modified code is then compiled into assembly code and code augmented for timing purposes on the assembly language. Because each shared reference is replaced with a procedure call, the object code resulting from compiler optimizations is substantially changed from its original form. This can cause a substantial accuracy problem in the instruction timing and cache behavior.

To deal with this problem, Trojan augments memory reference instructions with a special simulation call at the assembly-code level. After compiling the original application into the assembly code, the Trojan’s augmentation program inserts calls for memory-related instructions. Proteus uses the low level augmentation just for timing, but Trojan uses the augmentation also for memory references as well as for timing.

Figure 6 shows an example of the augmentation for a memory reference, as it is done on a Sparc architecture platform. In this example, the first instruction (add) does not access the memory, so it is directly executed on the host machine without having to call the simulator. Since the second instruction (ld) is a memory load, Trojan inserts extra instructions, which ends with an explicit call to the simulator (mem_issue). After receiving control, the simulator does some housekeeping work such as scheduling the next event; eventually, the control will return to the next instruction in the original program (nop).

Some simulators do not provide a way to simulate private memory references. That is, a private memory reference is assumed to always hit in the cache. Although the effects of these references are not explicitly visible to the other nodes, they indirectly affect the behavior of other nodes through the cache coherency protocol. For example, a private reference can displace a cache line used by a previous shared memory reference. If private variables are not simulated, there is no way to assess the effects of conflicting cache misses between private and shared memory references. In Trojan, all memory references are augmented and this enables Trojan to simulate private memory references as well as shared memory, resulting in more accurate simulation. Finally, if the user wants, there is an option in Trojan to ignore the simulation of private variables at simulation time.

6.2. Extra Functionality

Trojan is based on MIT Proteus, exploits its good features and increases its usability, accuracy, and flexibility by solving some of its original limitations. In addition to the functionality described above, Trojan:

- Supports relaxed memory consistency and sequential memory consistency. Proteus only supports sequential consistency.
- Improves simulation time by using a sleep-wakeup mechanism on cache misses and page faults.
- Supports spin and queue-based locks. On synchronization operations such as locks and barriers, Proteus supports only spin locks. Trojan adds flexibility by supporting also queue-based operations like those supported by DASH [4].
- Supports a full map ($Di r_{n} \times \overline{NB}$) [4] and a LimitLESS cache coherence protocol [9]. Proteus only supports the LimitLESS protocol.
- Simulates the effect of latency tolerating techniques such as prefetching.

7. Performance

In this section, we discuss the performance of Trojan with and without virtual memory. Table 2 shows the programs and input data sets used in the experiments. The programs were taken from the Stanford SPLASH (Cholesky)
<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Input Data Set</th>
<th>Size (lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky</td>
<td>Sparse matrix cholesky factorization</td>
<td>bcsstk14</td>
<td>1888</td>
</tr>
<tr>
<td>FFT</td>
<td>1D fast Fourier transformation</td>
<td>2**18 data points</td>
<td>1060</td>
</tr>
<tr>
<td>LU</td>
<td>LU decomposition</td>
<td>256 x 256</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 2. Application Characteristics

or SPLASH2 (LU and FFT) suites. All memory references (i.e., private memory references as well as shared memory) were simulated and queue-based operations were used for synchronization. To calculate network delays, Agarwal’s network model [1] was used, instead of simulating contention at each hop. This is done by computing the arrival time at the target node using the analytical model. Doing this significantly decreases simulation time without impacting accuracy.

7.1. Performance Without Virtual Memory

Trojan is based on MIT Proteus which is two orders of magnitude faster than Tango. Thus the performance of Trojan is basically the same as that of Proteus except for some extra features we added. Our original intent was to compare the performance of Trojan with that of Proteus. Unfortunately, however, this goal turned out very difficult since Proteus requires applications to be changed as discussed in Section 6. Hence we only show Trojan’s performance.

Table 3 shows the performance results on the various benchmark applications without supporting virtual memory. The table shows execution time (of simulated machine), simulation time, the number of target cycles simulated per second (rate), and slowdown 3 (number of simulation cycles needed to execute a single simulated cycles of a single process/thread). The number of processors varies from 1 to 64. The execution time is given to allow comparison with the results for virtual memory.

Several interesting observations can be made from Table 3. First, the slowdown factor varies from one program to another (although the variation tends to be small) from a factor of 50 to 450. This difference is a direct result of the different frequencies at which the applications interact with the simulator. Although in Trojan most instructions are directly executed and the context switching is fast (10 to 20 cycles), the slowdows in most cases are larger than 200, resulting from several extra overheads, which come from the scheduling mechanism within the simulator, the simulation of network, cache and memory modules, and the gathering of statistics.

Another interesting observation is that as the number of processors is increased, the slowdown decreases. This is due to the fact that the same input data is used for all simulations of each benchmark. This means that as the number of processors is increased, the smaller size of data set is allocated to each processor, which increases data locality. Increased data locality results in less need for network and memory simulations.

An alternative explanation for the slowdown trend is that synchronization operations are implemented and simulated with queue-based (not spin lock) operations as in DASH [4]. That is, when a processor issues a lock that another processor already holds, the processor is put to sleep and is waken up when it the lock becomes available. Thus the simulation cost for synchronization is amortized over a large number of processors which gives lower slowdown factors.

Finally, the results show that the performance for process-based benchmarks is not much different from that for thread-based. The average slowdowns for Cholesky (process-based), LU (thread-based) and FFT (thread-based) are 280, 160 and 349, respectively. If the Cholesky program would have been simulated using processes in host, the slowdown would have been much worse mainly due to the high overhead of context switching. This shows that a copy-on-write scheme used to emulate process-based applications performs very well.

To summarize, first, the average slowdown for the three benchmarks is around 250, which allows Trojan to simulate large data sets. Second, as the number of processors is increased, the slowdown goes down, which is good for simulating larger number of processors. Third, Trojan can simulate the two execution models efficiently without any additional performance loss.

7.2. Performance with Virtual Memory

Table 4 summarizes the performance of Trojan with virtual memory enabled. The table confirms most of the observations discussed above when virtual memory was ignored. However, the results show certain patterns that affect only virtual memory. First, the average slowdown without virtual memory for the three benchmarks was 262. With virtual memory the average slowdown is 271. As without virtual memory, this slowdown decreases as the number of proces-

---

3Slowdown is calculated by the equation \( \text{slowdown} = \frac{\text{simulation time}}{\text{execution time}} \) (not in cycles, but in seconds), assuming a 33MHz clock.
<table>
<thead>
<tr>
<th>Program</th>
<th>Procs</th>
<th>Exec. time (1000 cyc)</th>
<th>Sim. time (sec)</th>
<th>Rate (cyc/sec)</th>
<th>Slowdown (host/target)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky</td>
<td>1</td>
<td>58,843</td>
<td>765</td>
<td>76,880</td>
<td>439</td>
</tr>
<tr>
<td>Cholesky</td>
<td>4</td>
<td>22,325</td>
<td>860</td>
<td>103,806</td>
<td>321</td>
</tr>
<tr>
<td>Cholesky</td>
<td>16</td>
<td>10,673</td>
<td>1,444</td>
<td>118,229</td>
<td>282</td>
</tr>
<tr>
<td>Cholesky</td>
<td>64</td>
<td>9,115</td>
<td>1,406</td>
<td>414,781</td>
<td>80</td>
</tr>
<tr>
<td>LU</td>
<td>1</td>
<td>172,082</td>
<td>1,682</td>
<td>102,270</td>
<td>326</td>
</tr>
<tr>
<td>LU</td>
<td>4</td>
<td>91,680</td>
<td>1,667</td>
<td>219,951</td>
<td>152</td>
</tr>
<tr>
<td>LU</td>
<td>16</td>
<td>34,389</td>
<td>1,696</td>
<td>324,415</td>
<td>103</td>
</tr>
<tr>
<td>LU</td>
<td>64</td>
<td>17,062</td>
<td>1,911</td>
<td>571,184</td>
<td>58</td>
</tr>
<tr>
<td>FFT</td>
<td>1</td>
<td>61,587</td>
<td>825</td>
<td>74,571</td>
<td>451</td>
</tr>
<tr>
<td>FFT</td>
<td>4</td>
<td>21,112</td>
<td>844</td>
<td>99,944</td>
<td>335</td>
</tr>
<tr>
<td>FFT</td>
<td>16</td>
<td>6,870</td>
<td>954</td>
<td>115,118</td>
<td>331</td>
</tr>
<tr>
<td>FFT</td>
<td>64</td>
<td>2,992</td>
<td>1,066</td>
<td>179,552</td>
<td>277</td>
</tr>
</tbody>
</table>

Table 3. Performance of Trojan without Virtual Memory

<table>
<thead>
<tr>
<th>Program</th>
<th>Procs</th>
<th>Exec. time (1000 cyc)</th>
<th>Sim. time (sec)</th>
<th>Rate (cyc/sec)</th>
<th>Slowdown (host/target)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky</td>
<td>1</td>
<td>59,047</td>
<td>859</td>
<td>68,711</td>
<td>485</td>
</tr>
<tr>
<td>Cholesky</td>
<td>4</td>
<td>37,957</td>
<td>1,428</td>
<td>106,269</td>
<td>321</td>
</tr>
<tr>
<td>Cholesky</td>
<td>16</td>
<td>21,847</td>
<td>2,618</td>
<td>133,519</td>
<td>259</td>
</tr>
<tr>
<td>Cholesky</td>
<td>64</td>
<td>10,633</td>
<td>2,183</td>
<td>311,700</td>
<td>113</td>
</tr>
<tr>
<td>LU</td>
<td>1</td>
<td>172,576</td>
<td>2,240</td>
<td>77,021</td>
<td>434</td>
</tr>
<tr>
<td>LU</td>
<td>4</td>
<td>89,247</td>
<td>1,869</td>
<td>190,949</td>
<td>175</td>
</tr>
<tr>
<td>LU</td>
<td>16</td>
<td>29,169</td>
<td>1,713</td>
<td>272,290</td>
<td>123</td>
</tr>
<tr>
<td>LU</td>
<td>64</td>
<td>11,782</td>
<td>1,787</td>
<td>421,813</td>
<td>84</td>
</tr>
<tr>
<td>FFT</td>
<td>1</td>
<td>66,933</td>
<td>917</td>
<td>72,923</td>
<td>463</td>
</tr>
<tr>
<td>FFT</td>
<td>4</td>
<td>24,328</td>
<td>967</td>
<td>100,537</td>
<td>336</td>
</tr>
<tr>
<td>FFT</td>
<td>16</td>
<td>9,592</td>
<td>1,099</td>
<td>139,645</td>
<td>254</td>
</tr>
<tr>
<td>FFT</td>
<td>64</td>
<td>3,761</td>
<td>1,216</td>
<td>197,895</td>
<td>211</td>
</tr>
</tbody>
</table>

Table 4. Performance of Trojan with Virtual Memory

Second, in most cases, the programs took longer (in execution time) with virtual memory, mainly due to TLB misses. In a couple of cases, however, programs took less time with virtual memory. For example, LU program for 64 processors took 17,062K cycles without virtual memory. With virtual memory, it took 11,782K cycles, a 30% reduction. The main reason for this is due to the physical addressing of the cache. With physical addressing, cache behavior can be different from that of virtual addressing. This shows that virtual memory simulation is important when evaluating cache behavior.

Additional memory requirement for virtual memory was less than 5% of the total memory required. Specifically, without virtual memory, the amount of memory consumed by Cholesky using 64 processors was 44,876K bytes. When virtual memory was enabled, the total memory increased to 45,092K bytes. This additional 216K bytes were used for page table and TLB management, plus some other data structures.

8. Conclusion

Trojan is based on MIT Proteus and exploits its advantages: it delivers high accuracy through execution-driven simulation and high speed by running in a single address space and by directly executing each target instruction whenever possible. Trojan increases usability and accuracy by adding extra functionality. It supports process-model based benchmarks using a threads package and does not require applications to be modified in order to be simulated. In addition, Trojan simulates all memory references and exhibits
good speed by using sleep-wakeup mechanism for cache misses (and page faults). Trojan extends the flexibility of Proteus by allowing users to select various options such as relaxed memory consistency and sequential consistency. Finally, as virtual memory becomes more commonplace in parallel machines, Trojan supports it as an option with an additional 10% time and 5% memory overhead.

We have used Trojan to conduct many experiments on cache behavior, network traffic patterns, and application performance. Trojan has also been used for the validation of analytical machine models and to assess the performance of novel architectural organizations such as the use of networks of workstations working as a parallel machine. Specifically, without virtual memory, it would not be possible to simulate networks of workstations (NOWs) accurately.

There is a room for future improvement within Trojan. Though Trojan provides virtual memory simulation, it does not support multiprogramming (and scheduling). To reflect realistic behavior of applications in NOWs, the simulation of multiprogramming is needed. Future version of Trojan will provide some form of multiprogramming.

Acknowledgments

We would like to thank Eric Brewer for providing Proteus and for answering all our questions, and Jacqueline Chame, Weihua Mao, and Sungdo Moon for giving comments on a previous draft.

References


