The Design of COMPASS: An Execution Driven Simulator for Commercial Applications Running on Shared Memory Multiprocessors

Ashwini K. Nanda, Yiming Hu[*], Moriyoshi Ohara[**], Caroline D. Benveniste,
Mark E. Giampapa and Maged Michael
IBM T.J. Watson Research Center
P.O. Box 218 Yorktown Heights, NY 10598
{ashwini,ohara,cben,giampap,michael}@watson.ibm.com

* Currently at University of Rhode Island ** Currently at IBM Tokyo Research Lab

Abstract

Although shared memory multiprocessors are becoming increasingly popular in the commercial market place, the applications used to evaluate such systems in both academia and industry are still predominantly technical applications such as the Stanford SPLASH2[1] benchmarks. The difficulty in using commercial parallel shared memory applications such as transaction processing, decision support and web server applications has been in simulating the operating systems functions that are heavily used by these applications. In this paper we describe the design of an execution driven simulation tool called COMPASS (COMmercial PArallel Shared memory Simulator). We have used COMPASS at IBM to study the behavior of decision support applications and are currently studying the behavior of transaction processing applications and web servers.

1 Introduction

Shared memory multiprocessors are becoming increasingly popular as server platforms for commercial applications such as transaction processing, decision support and web servers. Therefore, it has become desirable to study the performance impact of these commercial applications on the shared memory server platforms in order to make the right design decisions. However, most of the contemporary simulators used to evaluate shared memory multiprocessors both in industry and academia are not suitable for running commercial parallel programs. As a result most architecture studies are confined to scientific applications such as the SPLASH2 benchmarks[1].

Scientific applications on shared memory machines usually spend very little time in the operating systems. Therefore, not simulating any OS activity does not result in any significant loss of accuracy for these applications. However, many commercial applications heavily depend on operating system services, and some of the applications spend a significant portion of their CPU time in the operating systems. This is because commercial applications tend to use sophisiticated inter-process communication mechanisms and I/O functions that operating systems provide. For example, our profiling data show that on both AIX[1] and Windows NT systems, Web Servers spend 70-85% of their CPU times on OS kernels. On-Line-Transaction-Processing (OLTP) applications such as TPCC[2] and decision support applications such as TPCD[3] spend about 20% of their time in the operating systems. These applications, moreover, generate a significant amount of I/O activity. Therefore, in order to study commercial application behavior with reasonable accuracy, one has to simulate the OS functions where these applications spend a significant amount of their execution time.

Because of the dependency on operating system services, currently available execution-driven simulators are not adequate for our purposes. Some simulators (e.g. Augmint[4] and Tango-lite[8]) require a user-level thread programming model, which demands changing applications that run on UNIX-based shared-memory machines. The user-level thread model may be ideal for relatively simple scientific applications because it is easy to port such applications to the thread model and, more importantly, because it is easy to simulate the thread model in a very efficient manner. Using the user-level thread model, however, is not a viable option for us because real-life commercial applications such as IBM DB2[5] typically use an OS-level process model and are extremely complex. It is not feasible to modify such applications to use a user-level thread model in a timely fashion. Some other execution-driven simulators support a process model. MINT[6] and Proteus[7] simulate multiple application processes within a single UNIX process by mapping multiple simulated process spaces to a single process space and by providing custom-made OS functions that support inter-process communication for simulated application processes. This approach, however, is not attractive for our purposes because it requires us to rewrite all of the OS functions that use process-associated data structures such as all of the inter-process communication functions and file I/Os. It is not feasible to rewrite these OS functions in a timely fashion. Moreover, majority of existing operating systems provide a 32-bit virtual address space for each process. Since all of the application processes share a single 32-bit space in MINT, each process can use only a small virtual space. This can be a significant limitation when we simulate a large-scale parallel machine with a large parallel application. Tango[8] also supports process-model applications and is similar to COMPASS in a sense that both simulators assign each simulated application process to a UNIX process. Tango, however, supports only applications written with ANL macros and does not simulate OS functions that affect the performance of commercial applications significantly.

1. Unix is a registered trademark of X/Open Company Limited. Windows NT is a trademark of Microsoft Corporation. TPCC and TPCD are trademarks of the Transaction Processing Performance Council. AIX and DB2 are trademarks or registered trademarks of IBM Corporation.

SimOS[9] is by far the most accurate simulator for most scientific and commercial applications. SimOS simulates computer hardware in enough detail to run an *entire* operating system. In fact, real-world applications can run on SimOS without modification. While SimOS is the most accurate simulation environment so far, we observe some potential problems that might limit its usefulness for our study. First, the current release of SimOS is based on SGI IRIX 5.2 operating system, which may not scale well beyond 8 or 16 processors. Porting another industrial or research OS to SimOS requires a major effort beyond our time and budget limitations. Second, SimOS uses about one fourth of the total user-accessible process address space for the SimOS and target OS code and data. While this causes no problem for most applications, it does result in an address conflict problem for DB2, which requires control of almost the entire address space.

We have developed COMPASS (COMmercial PArallel Shared Memory Simulator) keeping these requirements and constraints in mind. COMPASS uses the basic instrumentation and execution-driven simulation mechanism in a PowerPC version of the Augmint simulator[4]. We carefully designed mechanisms to simulate only important OS functions that affect the performance of our target applications, yet to support virtually all of the OS functions. This should be an ideal simulation environment for many commercial applications because, as discussed later, they often use a large number of OS functions and only a small subset of the used OS functions impacts the performance of the application substantially. To support a large variety of OS functions without modeling them, we decided to use a UNIX process for each simulated application process. This approach causes a process context switch for each simulated memory operation and slows down the simulation as a previous study [6] indicated. On an SMP system, however, the backend process and a frontend process can run on two different processors, and sending an event from the frontend to the backend will not cause a context switch. This significantly reduces the simulation overhead. In fact, that approach makes COMPASS very attractive for commercial applications and architecture studies because a large class of commercial applications can be ported to the environment without significant effort. Currently we can run the DB2 database server for TPC-C and TPC-D benchmarks and the Apache web server for the SPECWeb96 benchmark. We believe that the COMPASS approach of modeling the OS functions or algorithms that are important for a new application is more modular than the SimOS approach. COMPASS has to worry about the new function's compatibility only with a handful of previously modeled OS functions rather than with a complete OS like in SimOS.

The rest of the paper is organized as follows. Section 2 describes the structure of COMPASS. Section 3 explains the mechanisms used to simulate various OS functions and physical devices. Section 4 summarizes our experience in porting three important commercial applications to COMPASS. Section 5 briefly describes the use of COMPASS in architecture performance studies and Section 6 concludes the paper.

2 The Structure of COMPASS

The structure of COMPASS is shown in Figure 1. The COMPASS simulation environment consists of several *frontend processes* and a *backend simulation process.* The frontend processes constitute of several *application processes* that represent the target application and an *OS Server process* that simulates selected functions of the AIX operating systems.

The frontend processes are built by compiling the application source code to generate assembly code. The assembly code is then run through an instrumentation program which inserts special assembly code at end of each basic block and each memory reference. The inserted code calculates the timing information of the process by using the estimated execution time of each instruction based on the specifications of the microprocessor instruction set, assuming 100% instruction cache hits. For each memory reference, the inserted code also fills out an event data structure at run time with information on the reference type, the effective address, the reference size, and the cycle time at which the reference is generated. The data structure is passed to the backend simulation process through the event port (see Figure 2).

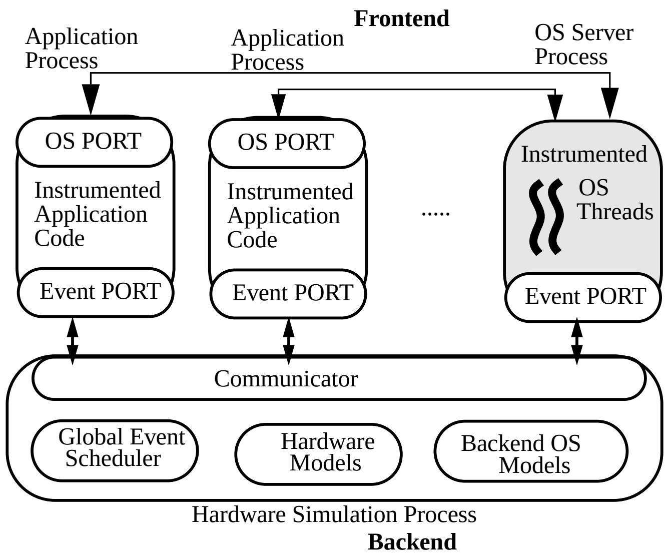

Figure 1: Structure of COMPASS

The operating system functions are selectively modeled in COMPASS. Important OS functions *directly* affecting the application performance are modeled using the *OS Server process.* For these OS functions, the memory reference behavior within the OS significantly affects the performance of the application and needs to be modeled. The application processes communicate with the OS server process through the *OS port* attached to each of them (see Figure 2). The OS server is executed in the user mode and is instrumented in the same way as the application processes. Other OS functions that *indirectly* affect the application performance are modeled as a part of the backend simulation process. For these OS functions, the memory reference behavior within the OS does not significantly affect the performance of the application but the backend needs to model the OS function so that the application runs correctly. The next section discusses more details about the OS modeling mechanism.

Contained inside each application process, the *event port* is responsible for communicating with the backend through a shared-memory communication interface called *communicator.* The event port also contains the per-process and per-event data structures which are shared between the frontend and backend processes. When the event port is invoked, it notifies the backend that it has a message, and in the normal case waits for a reply, which prevents the frontend process from proceeding. When the event information is received by the backend, the backend creates a task and inserts it in the *global event scheduler* with a time stamp indicating at which global simulation cycle the task is to be dispatched. When the task reaches the top of the global event queue, the function associated with that task is performed. Functions may cause additional tasks to be generated and placed in the global event queue. When all the tasks associated with a particular event have completed, the backend process replies to the frontend process, allowing it to proceed.

In COMPASS, the *Communicator* provides the interface between the frontend application processes and the backend simulation process. To reduce communication overhead to a minimum, this interface uses custom built Shared Memory Message Passing incorporating a shared memory segment and a set of blocking and non-blocking message passing primitives. To obtain optimal performance, the vast majority of communication between the frontend and backend processes is implicit, handled directly by loads and stores to the various data structures in the communicator's shared memory by instructions inserted into the frontend processes and the backend simulation process.

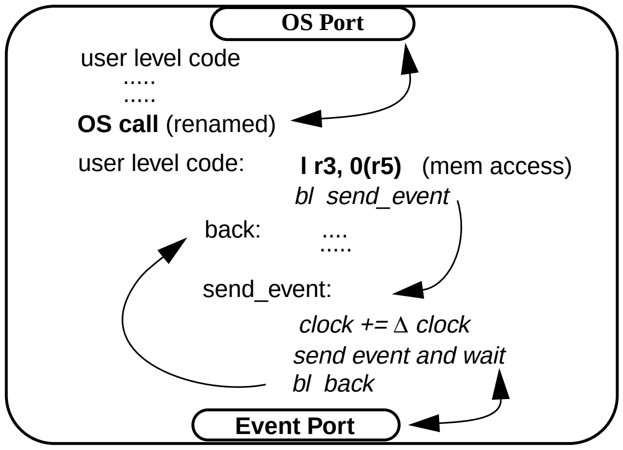

Figure 2: Communication with OS port and Event Port

Another important function of the communicator is to synchronize the frontend processes to simulate instruction interleaving. In COMPASS, each frontend process runs independently. When COMPASS is running on a single CPU or a small-scale SMP system, these frontend processes time-share the CPU resource. For the sake of simulation accuracy, the instructions of the frontend processes must be interleaved in a way as if each of the running process is executing on a separate CPU, resulting in a very fine-grained instruction interleaving. While it is possible to simulate this kind of fine-grained interleaving by forcing a context switch after each frontend instruction, doing so will result in an intolerable slowdown of simulation.

COMPASS uses a novel technique to solve this problem. Each frontend process maintains an ``execution time" value, which is the accumulated execution time of the process in CPU cycles. The value is within the process event port, which in turn is located in a shared memory section. For each instruction that generates an event (i.e, a memory-reference instruction or a synchronization instruction), and for each basic block, the instrumentation program generates instructions to update the execution time. If a process has a smaller time value than that of another process, the former is running ``behind" the latter. When a frontend process sends an event to the backend, it blocks and waits for a reply from the backend. Meanwhile, the communicator of the backend keeps scanning the event ports of all running frontend processes. It only picks up an event generated by the frontend process with the smallest execution time value. When the event is processed and replied, the slowest process can proceed to catch up with other processes blocking on event ports.

This approach enables COMPASS to simulate the instruction interleaving at the basic-block level, which is reasonably fine-grained. Meanwhile it minimizes the overhead of context switches, giving COMPASS an acceptable execution speed. Additionally, Implementing the application, OS server and the architecture models as separate processes enables COMPASS to run more efficiently on a parallel machine.

The backend simulation process simulates the target shared memory multiprocessor architecture including several levels of caches, memory buses, memory controllers, coherence controllers, network, and physical devices of the target computer system. The complexity of the backend architecture models depends on the amount of details being simulated. The simplest backend consists of only a one-level cache per processor and the most complex backend models all the other system components along with a two-level cache per processor. Running time of an application in the COMPASS environment depends heavily on the complexity of the backend models.

3 Simulation of OS functions and Physical Devices

As mentioned before, COMPASS models the OS functions selectively. We profiled the target commercial applications to determine the OS functions where they spend significant amount of time. Table 1 shows a summary of some profiling results for the TPCC and TPCD benchmarks on DB2 and the SPECWeb[10] benchmarks on the Apache web server[11] on a 4-way AIX/PowerPC SMP machine. The table shows the user and OS times as a percentage of the total CPU time which excludes wait time due to disk IO. This particular run of SPECWeb spends about 85% of the execution time in the OS. The interrupt handlers related to ethernet access and disk I/O access consume 37.8% of the CPU time. Out of the 47.3% kernel time, about 42% is spent in a handful of OS calls, such as, `kwritev, kreadv, select, statx, connect, open, close, naccept` and `send` which are predominantly due to the TCP/IP stack. Only 5.3% of the CPU time is spent in miscellaneous other calls.

For both TPCD and TPCC, the OS activities account for about 20% of the execution time. The interrupt handler time for these benchmarks are mainly due to disk I/O and the interval timer. The significant OS calls accounting for most of the OS time are `kwritev, kreadv, mmap, munmap` and `msync,` which are related to disk I/O and the file system. Therefore, it is imperative that we model the TCP/IP stack, disk I/O and the OS calls associated with them as well as the corresponding interrupt handlers accurately in order to obtain credible performance results for these Target benchmark programs.

Table 1: User vs. OS time

benchmark	user time	OS time		
		total	interrupt handlers	kernel
SPECWeb/ Apache	14.9%	85.1%	37.8%	47.3%
TPCD/DB2 (100MB DB)	81%	19%	8.6%	10.4%
TPCC/DB2 (400MB DB)	79%	21%	14.6%	6.4%

It is clear from the execution profiles that only a handful of the OS functions have significant *direct* impact on the execution time of the set of applications we considered. The memory access behavior of these OS functions have to be captured and simulated in order to get a fairly accurate estimate of the application execution time on the simulated target architecture. We call these OS functions *category 1* functions. The *category 1* functions are simulated using the *OS server* as described shortly. There are other important OS functions where an application may not be spending significant portions of its execution time, but which can impact the application performance *indirectly.* We call these OS functions *category 2* functions. Two examples of category 2 functions are process scheduling and memory management (including page movement in distributed memory systems). Both these functions can significantly impact the cache behavior and physical memory access latencies of the application. However, since modeling an OS function in the OS server requires relatively more effort, the category 2 functions are modeled in the backend simulator process and their *effect on the memory reference behavior* of the application processes is modeled accurately.

3.1 The OS Server

Under an actual execution environment the OS kernel runs in the kernel address space. If one process running in the kernel mode makes some changes to the kernel memory, such as allocating a disk buffer or an mbuf, generally another process running in the kernel mode should be able to see these changes. In other words, most of the kernel code executes in a shared memory environment. During simulation, however, the simulated kernel code must run in the user mode. As a result, we can not simply instrument kernel source code and link the instrumented code with the application, because in such a case each process will run the kernel code in a separate address space.

One possible solution is to change kernel source code such that all its data are in a shared memory section. By doing so, multiple processes can be linked with the modified kernel code. The modified kernel runs within the address space of each process, but the processes can see each other's changes via shared memory. This approach, however, requires carefully examining all the source code of the entire simulated kernel. Any non-local data must be moved to the shared memory section. This approach is tedious, and is error-prone. Moreover, many OS executable entities, such as interrupt handlers or the kernel thread for virtual memory garbage collection, are in the bottom half of kernel and do not have a process context. It is not easy to simulate the bottom half code using the above scheme.

COMPASS addresses this problem with a multi-threaded OS server using POSIX threads[12]. For a multi-process application, there is a one-to-one mapping between a user process and an OS thread running in the server. Each OS thread provides kernel services for its corresponding user process. The communication between the OS server threads and the user processes are through shared-memory IPC. Since multiple threads share the same address space, the address sharing problem of multiple kernel instances is solved. Moreover, dedicated threads can be scheduled to simulate bottom half kernel activities.

The OS server is a stand-alone, multi-threaded program that simulates *category 1* OS functions in COMPASS. Upon starting, the OS server spawns a pool of *OS threads*. Each OS thread has two IPC ports in the shared memory section, namely the *OS port* and the *event port*. The OS port is used to accept OS calls from an application process. The event port is for sending memory-reference events to the backend process. Initially all OS threads are said to be in the "single" state because they are not bound to any user process. Each thread monitors its own OS port, waiting for a *connection request* from a frontend process.

Each application process also has an event port and an OS port. When a new process is created, it first establishes a connection between itself and the backend via the event port. Later when the process is running in the user mode, all its memory-reference events are sent to the backend via the event port. The newly created process also sends an OS connection request to the OS server via the OS port. An OS thread will receive the request and bind itself to the frontend process. The OS thread is now said to be "paired" with the corresponding application process. Meanwhile, the application process also passes its own event port setting to the OS thread, so the latter can use the same event port of the former to communicate with the backend.

The COMPASS instrumentor replaces all OS calls in a user application with COMPASS OS stubs. When executing in the user mode, an application process sends events to the backend simulator via its event port. When an OS call is encountered, the OS stub for that OS call is executed. If the stub finds that the call can be handled by an OS server, it sends the OS request, along with its arguments, to its "companion" OS thread via the OS port. The application process then halts. When the OS thread receives an OS call, it invokes relevant kernel code to conduct the OS service. Since the kernel code executed in the OS server is also instrumented, the OS server process generates memory-reference events. These events are sent to the backend through the event port of the thread, which is the same event port of its companion application process. The OS thread returns the OS call by sending the result and/or the error code back to the application process after which the application process resumes execution.When the frontend process exits, it sends an EXIT message to its OS thread counterpart. The OS thread becomes "single" again and can accept an OS connection request from another new process.

The multi-threaded OS server approach is somehow similar to a micro-kernel operating system such as Mach[13]. This scheme enables us to simulate the kernel code in the user space. It clearly separates the OS code from user applications and other simulation infrastructures. In this way one can change the algorithms of the OS server, by trying, for example, different scheduling and locking schemes, without having to change other parts of the simulator. It is also easy to replace the current AIX OS server with a new OS server that simulates other operating systems. Moreover, the scheme also has good extensibility. When new OS services are to be supported, they can be added to the existing OS server, or can be added to a new OS server running in parallel with the old server(s). The stubs in the application side will direct the OS calls to the appropriate server. Finally, since the OS functions are simulated in a separate process, they will not occupy user address spaces, thus reducing the possibility of address conflicts between the simulator and user applications.

3.2 Interrupts and Traps

The ability to simulate interrupts and traps is vital for simulating many important OS activities, such as process scheduling, virtual memory management, as well as disk and network I/Os. When a device simulator in the backend raises an interrupt, there must be a way for the backend to stop a frond-end process, which is running in a separate address, to execute the interrupt handler for that interrupt. Posting a signal to the frontend process will not work, because signal handlers are generally called only during system calls or on a context switch.

COMPASS solves this problem by introducing a “CPU-states” data structure in the Communicator module (see Figure 1) using shared memory accessible by the application, the simulator and the OS server processes. Each CPU has an “interrupt request” flag bit as well as an “interrupt enable” bit in the CPU-states structure. When the backend schedules an interrupt for a given processor, the former sets the “interrupt request” flag bit in the CPU-state area of that processor. The frontend processes, on the other hand, are continuously running at this moment. When a process encounters a memory-reference instruction, an IPC subroutine is called to send the memory-access event to the backend via the event port. We let the frontend process check the interrupt request flag before returning from the IPC subroutine. If the flag is set and the processor is interrupt-enabled, the IPC subroutine calls the appropriate interrupt handler before it returns, effectively stopping the frontend process. When the interrupt handler returns, the frontend process can proceed for the next instruction. Since not all instructions in a frontend process generate memory events, the interrupt flag bit will not be checked until a memory-access instruction (or sometimes, a branch instruction) is encountered. This may cause an average delay of several instructions on interrupt responses. We believe that this small delay will not cause any problems, since interrupts are asynchronous events. On the other hand, the scheme can accurately simulate traps (such as page faults) caused by memory reference instructions, since the same instruction generating the trap will check the interrupt bit after the memory event is sent to the backend.

Interrupt handlers run in the bottom half of kernel, operating in the kernel address space. This implies that they must be invoked within the OS server during simulation. Consequently, there are two different ways to handle interrupts in COMPASS: one for applications running in the user mode, the other for the kernel mode.

When an application is running in the kernel mode, the actual instructions executed, and the memory-access events generated, are from an OS thread in the OS server. When the OS thread detects the setting of the interrupt flag, it can simply call the interrupt handler or schedule a kernel thread to serve the interrupt. The interrupt handler or the kernel thread runs in the same address space of the OS thread, which is the simulated kernel space.

The scenario is a little bit more complex for programs running in the user mode. When a user-mode process detects the setting of the interrupt flag, it can not invoke an interrupt handler directly, otherwise the interrupt handler will run in the user space instead of in the kernel space. Rather, the frontend process sends a “pseudo interrupt request” to its OS thread via the OS port. When the OS thread receives the pseudo interrupt request, it calls the interrupt handler of that interrupt. When the interrupt handler returns, the OS thread informs the frontend process to continue.

3.3 Modeling Category 2 OS functions in The Backend

The category 2 functions are modeled in the backend process. We do not simulate these functions in detail. We simply call the actual AIX system calls on the host machine to perform the real tasks corresponding to these functions so the program can execute correctly. However, we attempt to model the resulting effect of these functions on the application’s memory behavior fairly accurately. For example, if a system call involves a block data transfer, we generate the same number of memory request events to the backend architecture models to simulate the data transfer.

3.3.1 Virtual Memory Management

In process based parallel programs the different processes share data by using the operating system shared memory routines (e.g. *shmat*, *shmget*, *shmdt* in AIX). To support these functions we added the following capability to the simulator: when a process makes a call to a shared memory routine, a stub routine is executed in which the actual call to the shared memory routine is made, and a function in the simulator backend is invoked. When a call is made to *shmget*, this function will create a model for a common shared memory descriptor in the backend simulation process. This common shared memory descriptor links the Shared Memory Flag argument in *shmget* to a unique descriptor for that shared memory segment. This descriptor is common to all processes. When a call is made to *shmat*, page table entries are created in the page table model of the calling process. Each process has its own page table model, with page table entries for each shared page. In a separate structure in the backend we keep a hash table of the home nodes of each of the pages hashed by physical address. The home nodes can be assigned at the time of page creation (if a round-robin or block page placement policy is being used) or when the page is first referenced (if a first-touch page placement algorithm is used). When an address is passed to the simulator backend, it performs the virtual to physical address translation by checking the process’ page table for the appropriate address.

3.3.2 Process Scheduling

We needed a way to schedule the Application processes onto our virtual (simulated) processors. To do this we implemented a process scheduler in the simulator backend. This process scheduler keeps a mapping of processes and their associated processors. If there are more processes than processors in the system, then certain processes will not be assigned a processor, and that process will be blocked. When the simulator starts, it assigns processors to processes as long as there are free processors. All other processes are placed on a ready queue and wait for an available processors. Processors become available as the processes assigned to them execute blocking OS calls. At that time, the process scheduler will schedule a process on the ready queue to the newly-freed processor. When a process completes a blocking OS call it will be scheduled if there are free processors. Otherwise, it will be placed on the ready queue. We have implemented three different process schedulers. In the default or FCFS scheduler a process will be assigned the first available processor. In the optimized or affinity scheduler, if more than one processor is free, the process will try to choose a processor it has used before, preferably the one it was using before it was blocked. If it cannot find any processor it has used before, then it looks for processors on the same nodes as processors it used before. The third process scheduler is a pre-emptive scheduler which interrupts processes at certain intervals and assigns their processors to waiting processes (if any). The pre-emption interval can be changed in the simulator. The pre-emptive scheduler can be used with the default or optimized scheduler.

3.3.3 Blocking OS calls

Certain OS calls will cause the calling process to block while waiting for another process. Since we use actual AIX processes in our simulator, we need to perform additional functions when such blocking OS calls are encountered or our simulator may deadlock (e.g. the blocked process waits for another process but the backend simulator cannot run the process that is being waited for.) In the current simulator, when a blocking OS call is about to be executed, we execute a stub routine which in turn creates an event for the backend simulation process. In addition, this process is marked as being blocked in the operating system. In the backend, the process scheduler will be informed that the process has been blocked, and free up the processor on which it was running. Once the blocking OS call completes, another stub routine is called which causes an event to be scheduled in the backend. This event informs the process scheduler that the OS call has completed and that the process is now ready to be scheduled again.

3.4 Physical Devices

A fully functional simulation environment must provide some degree of support for physical device simulation, which is necessary for OS simulation. Currently we have implemented simulation models for three kinds of devices, namely the real time clock, the Ethernet and the hard disk drives.

4 Porting Applications to COMPASS

Since applications run on a real operating system in the COMPASS environment, porting applications to COMPASS is relatively straightforward. Unlike SimOS, however, COMPASS is not designed to simulate every instruction in the binary; we need to specify a part of the application program that we are interested in and want to simulate. More specifically, we need to follow the following four steps to run an application on COMPASS. First, we need to identify the source files to be simulated and then modify the build script to instrument those files. Second, we need to identify a part of the program for which we need to avoid simulating the memory behavior. For example, the current implementation of COMPASS does not support simulating memory accesses for signal handlers. Third, we need to rename OS calls that can cause deadlocks and to supply a stub library for those OS calls. Finally, we need to write code to simulate OS functions that can significantly affect the performance of the application.

In the following sections, we discuss our experiences in porting two important commercial applications - IBM DB2 database server and the public domain Apache Web server to COMPASS.

4.1 IBM DB2 Database Server

Porting DB2 to COMPASS was a qualitatively different experience from porting other applications (such as the SPLASH-2 programs), because of the size and the complexity of the program. The DB2 system is written in C++ and consists of a number of executables, shared libraries, and administrative tools. We decided to instrument one executable and one shared library to simulate all of the main DB2 server processes. Because instrumentation of the source code increases the size of the binary significantly, we needed to split the shared library into 4 libraries so that each instrumented library fits within the size limit of shared libraries in AIX 4.1.

Within the instrumented binaries, we needed to disable generating memory-access events from signal handlers and static constructors and deconstructors of C++ objects, for which the current implementation of COMPASS does not support the memory simulation. As described earlier, for each simulated process, we maintain a context record that includes a flag bit to control the generation of memory-access events. The augmented code checks the flag for each memory access and does not generate an event if the flag is zero. For signal handlers, we manage the control flag by using a non-augmented wrapper function that is installed as a signal handler for all signal events in DB2. Signals invoke the wrapper function that manages the control flag before and after the function calls the signal handler that DB2 provides. The source-code preprocessor of COMPASS automatically modifies the source code to install the wrapper function as a signal handler. For static constructors and deconstructors, we disable event generation by using a statically-initialized record for the process context. We took this approach because static constructors and deconstructors are called before and after main() is called.

4.2 Web Server Applications

Porting the Apache web server itself to COMPASS was a relatively simple task, because COMPASS has already provided most of the OS functions needed by Apache, and because Apache is a relatively simple program, with only about 38,000 lines of code. The necessary modifications to the source code are mainly for dealing with signals and time-out. A web server is driven by requests generated by web clients. We use SPECWeb96, a standardized Web server benchmark, to generate requests to Apache. SPECWeb96 consists of two parts: a file set generator and a workload generator. Before testing a web server, the file set generator must be run in the server machine to populate a test file set consisting of many files of different sizes. The workload generator then runs in one or several computers connected to the web server machine being tested via a TCP/IP network. It simulates web clients by sending HTTP ``GET” commands to the web server, requesting for files in the test file set.

However, when simulating Apache under COMPASS, we can not simply run the SPECWeb96 workload generator on one or several client machines and send requests to Apache under simulation. The main reason is that SPECWeb96 will simply time out and drop connections to the server, because the server under simulation is too slow. We solve this problem by generating an intermediate HTTP *request trace file* using the Apache web server driven by the SPECWeb96 benchmark. We then implement a trace player that reads the trace file and feeds the requests to a web server.

5 Simulation Slowdown

COMPASS is currently being used at IBM to study the interaction of three commercial applications, namely TPCC and TPCD on IBM's DB2 database server, and SPECWeb96 on the Apache web server with a variety of shared memory architectures such as CCNUMA, COMA and software DSM multiprocessors. Some of our early results on TPCD/DB2 obtained using COMPASS were reported in[14].

Table 2: Slowdown on uniprocessor(133 MHz PowerPC)

	Raw	Simple Backend	Complex Backend
execution time(secs)	52	16149	34841
slowdown	1	310	670

The slowdown of an application running on COMPASS compared to the raw execution time of the application on the same host depends mainly on three factors: (1) how much of the application code is actually instrumented for simulation, (2) the complexity of the target architecture model and (3) number of processors in the Host system. The amount of code to be instrumented is controlled by a Simulation ON/OFF switch. The ON/OFF switch can be inserted anywhere in the application (or OS server) code to selectively disable instrumentation of uninteresting parts of the code.

The raw execution time, simulation execution time and slowdown factor for a TPCD query on a 12MB database on a uniprocessor system as well as a 4-way SMP system are shown in Table 2 and Table 3 respectively. The simple backend architecture model simulates only a single level cache. The complex backend architecture model simulates a complete CCNUMA system. These slowdown factors are comparable to those in SimOS[9]. It is worth noting that COMPASS runs more than twice as fast on the SMP as on the uniprocessor for the complex backend (after properly scaling the execution times to the respective processor frequencies).

Table 3: Slowdown on a 4-way SMP (166 MHz PowerPC)

	Raw	Simple Backend	Complex Backend
execution time(secs)	23	9004	12650
slowdown	1	391	550

6 Conclusion

In this paper we presented the design of an execution driven simulator called COMmercial PArallel Shared memory Simulator (COMPASS). COMPASS is designed to run important commercial applications such as transaction processing, decision support, and web servers. These commercial applications spend substantial amount of their execution time on OS functions and therefore important OS functions have to be simulated in order to get reasonably accurate performance results for these applications. The important OS functions to be simulated are determined by profiling the target applications. COMPASS uses novel techniques to simulate these OS functions in the user mode. By doing so, COMPASS is able to close a gap in the performance evaluation of future scalable servers targeted for the commercial market.

7 References

[1] Steven Cameron Woo, Moriyoshi Ohara, Evan Torri, Jaswinder Pal Singh and Anoop Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," Proceedings of the 22nd International Symposium on Computer Architecture, June 1995.

[2] Transaction Processing Performance Council, "TPC Benchmark C, Standard SPecification Revision 3.3", April 8, 1997. Available on-line at http://www.tpc.org/cspec.html.

[3] Transaction Processing Performance Council. TPC Benchmark D: Standard Specification. 19 December 1995.

[4] Anthony-Trung Nguyen, Maged Michael, Arun Sharma and Josep Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In *Proceedings of the International Conference on Computer Design*, pp. 486-490, October 1996.

[5] IBM Corporation, DATABASE 2 Information and Concepts Guide for Common Servers Version 2, Document number S20H-4664-00, May 1995.

[6] Jack E. Veenstra and Robert Fowler, "MINT Tutorial and User Manual," Tech. Report 452-CCD-95-7, University of Rochester, June 1993.

[7] Eric A Brewer et. al., "Proteus: A High-Performance Parallel Architecture Simulator," MIT/LCS/TR-516, MIT, September 1991.

[8] S.A. Herrod, "Tango Lite: A Multiprocessor Simulation Environment," Technical Report, Stanford University, Computer Systems Laboratory, November 1993.

[9] Mendel Rosenblum et. al., "Complete Computer Simulation: The SimOS Approach," IEEE Parallel and Distributed Technology, Fall 1995.

[10] The Standard Performance Evaluation Corporation, "SPECWeb96 Benchmark," http://open.specbench.org/osg/web96/.

[11] The Apache Team, University of Illinois, "Apache HTTP Server Project," http://www.apache.org/

[12] Portable Operating System Interface (POSIX) Part I: System Application: Program Interface (API). IEEE/ANSI Standard 1003.1, 1996 Edition.

[13] David Golub et. al., "Unix as an Application Program," Proceedings of the USENIX Summer Conference, June 1990.

[14] Moriyoshi Ohara, Ashwini K. Nanda, Caroline Benveniste and Mark Giampapa, "Memory access characteristics of DB2/TPC-D on NUMA multiprocessors", 1997 Workshop on Performance Analysis and its Impact on Design (PAID'97, held in conjunction with ISCA'97), June 1997.