Reprinted from *International Journal in Computer Simulation*, Vol. 1, pp. 31-58 (1991)

The Efficient Simulation of Parallel Computer Systems†

R. G. Covington
Jet Propulsion Laboratory, MS 125-233
4800 Oak Grove Drive
Pasadena, CA 91109

S. Dwarkadas, J. R. Jump, and J. B. Sinclair
Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251

S. Madala
Coherent Systems, Inc.
1000 Bay Area Blvd., Suite 206
Houston, TX 77058

Abstract: -- An ongoing research project involves the design and evaluation of a software system for simulating parallel computers. A major goal in the development of this system was to avoid the high overhead associated with the conventional instruction-level simulation of sequential computers, but to retain the accuracy of that technique derived from its use of the execution of real programs. The resulting system is program-driven, but the overhead is significantly reduced by profiling the program to get timing estimates for its basic blocks, which are then used at run time to generate process execution times dynamically while avoiding a detailed emulation of each instruction's execution. A number of experiments dealing with message-passing computer systems have been performed in order to determine the level of accuracy that can be expected from its performance predictions and to measure its overhead.

Index Terms -- Architecture models, efficiency, parallel computers, parallel programs, performance, simulation, testbed, validation.

---

† This research was supported by Texas Instruments Grant No. TI720845RJ, NSF/ONR Grant No. N00014-87-K-0324, SDSU/SDIO Contract No. N66001-85-D-0203, NASA Grant No. NAG 0-208, and a Shell Doctoral Fellowship.

1. Introduction

One of the most significant trends in high performance computing is the development and widespread use of parallel processing systems. This has created a need for accurate and efficient tools to evaluate the performance of parallel systems under realistic workloads. These tools must allow the system designer as well as the programmer to study the interactions between hardware and software, with the end goal of obtaining a better match between a parallel computing system and the programs it executes. This paper describes the design and evaluation of a set of software simulation tools, called the Rice Parallel Processing Testbed (RPPT), developed for this purpose.

The conventional way to simulate the execution of programs on a computer is to use a host computer to emulate in detail the effect of executing instructions on a target computer. This approach is commonly referred to as *instruction-level simulation*. It has the advantage that very detailed information about the behavior of a proposed target architecture can be obtained. Programs written for the target processor can be executed with the instruction-level simulator to determine how they would behave if they were executed on the target processor. Moreover, timing information about the execution of programs on the target processor can be obtained to any required degree of accuracy by increasing the detail in which the individual instructions of the target processor are emulated. The main disadvantage of this type of simulation is that a number of host instructions must be executed to emulate a single target instruction, and this overhead increases with the level of emulation detail that is implemented. It is not uncommon to use hundreds of host instructions just to emulate one target instruction [1].

When we consider the simulation of parallel computing systems composed of several interacting processors, the amount of overhead that would be incurred with instruction-level simulation can become intolerable. Not only does the number of instructions that must be emulated increase with the number of processors, but the time required to simulate the interconnection structure used to provide communication paths between the processors can become a significant additional factor. Parallel computing systems with over 100 nontrivial processors are currently available, and systems with over 1000 are proposed. Although instruction-level simulation has been used for parallel systems [2, 3], there is a need for simulation techniques that can produce accurate performance predictions without the prohibitively large overhead of instruction-level simulation.

One way to reduce the complexity of instruction-level simulation of parallel computers is to use probabilistic models to represent the behavior of a program [4]. We call this approach *distribution-driven simulation*. It eliminates entirely the high cost of emulating target processor instructions. However, since the simulation is driven by a statistical approximation of program behavior instead of by the execution of an actual program, the approach can not be used to predict the detailed execution behavior of a system and is usually limited to high-level comparisons of different architectures.

Another technique that has been effectively used to study uniprocessor architectures is *trace-driven simulation*. First, a program is executed on a computer that has the same instruction set as the target processor, and a trace of program execution information (the event record) is collected. The trace data is then used to drive a simulation of the proposed architecture. This can be a highly accurate form of simulation since the trace data is derived from an actual execution of a program as in the instruction-driven approach. However, it can only be used to simulate architectures that are similar to the one used to produce the trace. This technique has been most useful in the study of memory organizations, especially cache memories. Unfortunately the technique does not extend well to the simulation of parallel computers even if all the processors are the same as the one used to generate the trace. Although there has been recent work in the efficient parallel simulation of multiprocessor caches using address traces[5][6], one problem is that variations in timings and delays in the system can affect the ordering of events and even which events occur, so that the traces are no longer valid [7].

Execution-driven simulation is a relatively new simulation technique that can be used to produce performance predictions that are nearly as accurate as those obtained with instruction-level simulation at a fraction of the overhead of that method [8]. It combines some of the properties of trace-driven and instruction-level simulation. As with both of these, it is program-driven in that the execution of a real program is used to drive the simulation. However, unlike instruction-level simulation where the execution of each individual target machine language instruction is emulated by the execution of many host instructions (even when the host and target machines are identical or nearly so), execution-driven simulation simply executes the machine language instructions of the program directly on the simulation host. Similar techniques based on the basic concept exploited by execution-driven simulation have also been proposed in references [7, 9, 10, 11].

A principle disadvantage of execution-driven simulation is that some of the timing details of instruction execution are lost since individual instructions are not emulated. As a result, the time required to execute a sequence of instructions can not be predicted as accurately as with instruction-level simulation. However, the errors due to this are usually quite small. Therefore, execution-driven simulation offers a compromise between the high cost and accuracy of instruction-level simulation and the speed but relatively lower accuracy of distribution-driven simulation.

This paper reports on recent research aimed at evaluating the effectiveness of execution-driven simulation. The goal of this work was to develop a complete simulation testbed based on this technique and to perform experiments that would provide a quantitative evaluation of its accuracy and efficiency. A major result of this effort is the *Rice Parallel Processing Testbed (RPPT)*, a sophisticated and powerful simulation system that can be used to investigate a number performance issues concerning the execution of concurrent programs on parallel computing systems [8]. RPPT simulation models for architectures can be written at different levels of detail, permitting the user to trade off accuracy against computational efficiency. Since the architecture models are driven by the executions of real programs, data-dependent computation times, interprocess communications and synchronization delays, and other architecture-specific timing information can be accounted for with a high degree of accuracy.

An important feature of the RPPT is its flexibility. Since the architecture models and the parallel programs are decoupled, an algorithm designer can test several different algorithms on a fixed architecture model. It is also possible to study the performance of a single algorithm on systems with fundamentally different basic architectures (e.g., a shared bus multiprocessor *vs.* a multiprocessor with a multistage interconnection network or a hypercube). This is especially important in the study of new concurrent architectures, particularly those intended for specific and/or dedicated applications.

Since the major advantage claimed for execution-driven simulation compared to instruction-level simulation is that it is much more efficient with only a small loss in accuracy, it is important to evaluate this claim. Towards this end, we have conducted a number of experiments with the RPPT to measure its accuracy and efficiency. The efficiency has been investigated by first measuring the time required to simulate (on a single processor) the execution of a parallel program on a parallel system. Then, that time is compared to the time required to simply execute the same program (in a pseudo-concurrent environment) on the testbed host. The ratio of the simulation time to the execution time, called the slowdown ratio, gives us an indication of the amount of overhead introduced by the simulator. To investigate the accuracy of the RPPT, we have measured the time required to execute parallel programs algorithms on a real parallel system and compared that with the times predicted by the RPPT simulating those programs on a simulation model of the parallel system.

The next section presents a definition and description of the execution-driven technique, including the process of profiling used to insert timing instructions into the parallel program used to drive the simulation. Section 3 describes the organization and use of the RPPT. The results of experiments used to validate the performance and measure the efficiency of the RPPT are given in Section 4.

2. Execution-Driven Simulation

In execution-driven simulation, execution of the program and the simulation model for the architecture are interleaved. That is, first a process is selected and executed until a process interaction point is encountered. A *process interaction point* is a state in a process' execution at which its immediate future behavior and/or timing potentially depends on the actions of some other process(es). Examples of process interaction points include sending and receiving messages in distributed memory systems and access to shared variables in shared memory multiprocessors. Second, any data movements between processors due to each interaction are simulated and simulation time is advanced. These two steps are repeated, servicing all processes, until every process terminates or the simulation is otherwise stopped. A major advantage of this approach is that there is almost no overhead required to simulate the activity within a processor; only the interactions between processors must be simulated in detail. In other words, the logical process activity internal to a single processor is accounted for by simply executing those parts of the process on the host computer. Overhead due to the need to maintain an event list and to simulate data movement is only encountered at those points where processes must interact.

Fig. 1 illustrates execution-driven simulation and how it differs from instruction-level simulation. The term *basic block* used in the third box in the figure refers to a sequence of assembly language instructions such that all instructions in the sequence are executed in order exactly once every time the block is entered, and the sequence is not contained in any larger basic block. Basic blocks always begin at a labeled instruction or immediately following a branch instruction and terminate at a branch instruction or just before a labeled instruction. Basic blocks are defined for HLL source programs in an analogous way.

The boxes marked with asterisks are where the two simulation techniques differ. Instruction-level simulation replaces "execute" by "emulate" and "basic block" by "instruction" as indicated by the parenthesized words in these two boxes. The first box shows how an instruction-level simulator performs a detailed emulation of each instruction, while execution-driven simulation simply executes instructions until the end of a basic block is reached. The emulation of a single target instruction requires the execution of several host instructions (100 or more for complex processors). This accounts for the significant reduction of simulation overhead in execution-driven simulation relative to instruction-level simulation.

For instruction-level simulation, there is no overhead associated with an instruction that is not labeled, is not a branch instruction, and does not require interaction with another process. Those instructions that branch or are labeled incur a small overhead to increment the cycle count. Only when the simulated process needs to interact with another process is a significant overhead incurred. At that time, an event queue insertion must be processed and the user-supplied program that defines the interconnection network structure must be executed.

The way elapsed time is accumulated between process interaction points by the cycle count variable accounts for the high accuracy of execution-driven simulation. It guarantees that the conditional branches of a program are evaluated and executed, so that all of the data-dependent properties of a program are taken into account during a simulation. This gives rise to a very accurate representation of the system's workload, where all process interaction points occur in the proper sequence and at the proper simulation time.

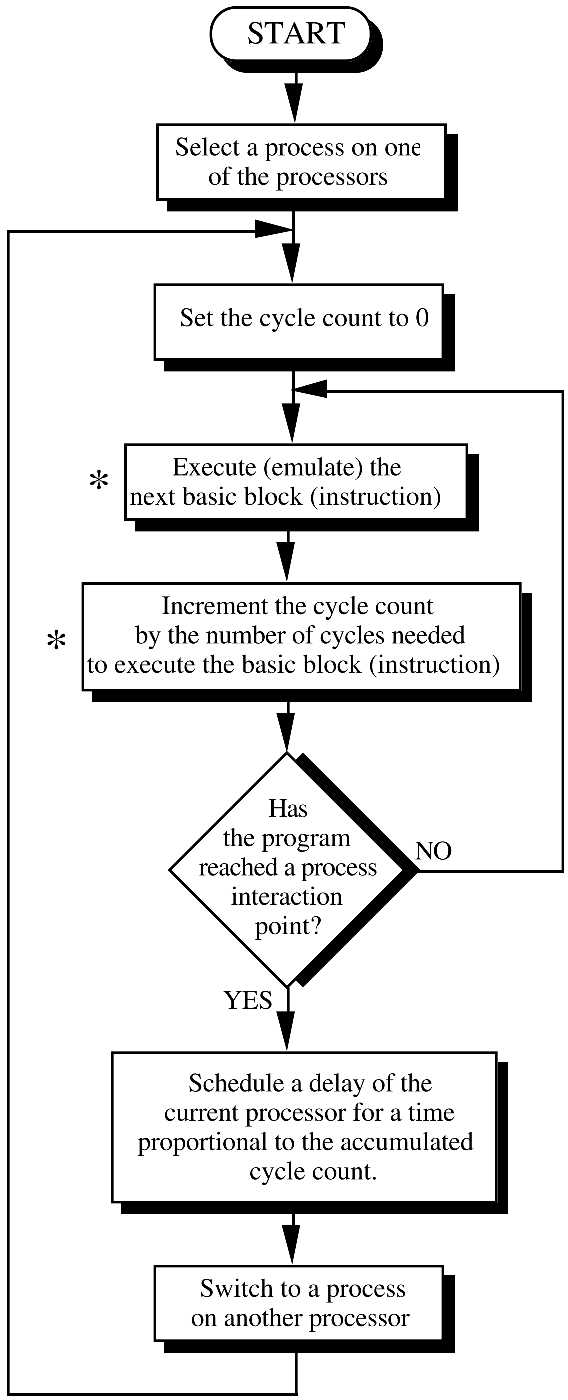

Fig. 1. Execution-driven and instruction-level simulation execution cycle.

Instructions that manipulate the cycle count variable are inserted at the start of each basic block of a source program by a utility program called a *profiler*. The profiler must perform two functions. (1) It detects all basic blocks in the assembly language source. (2) For each basic block, the profiler examines its instructions and uses information about instruction and memory-access timings to compute an estimate of the amount of time required to execute all of the instructions in the block. This estimate is then inserted into the source program in the form of a few instructions on each basic block boundary that increment the cycle count by the amount of the estimate for that block.

When the program is executed, the inserted instructions are executed as well, and they dynamically maintain a timing estimate for the sequence of basic block executions of the program. We call this *standalone* or *native profiling* because the modified assembly language program is executed directly on the processor for which the basic block timing estimates were obtained. Since the only run-time overhead occurs on basic block boundaries, this typically results in a relatively low slowdown ratio, usually 1.5 or less. The accuracy is limited only by the profiler's ability to estimate accurately the time required within each basic block. However, an important consequence of standalone profiling is that the instruction set of the target processor must be the same as the instruction set of the host processor.

Since it is not always possible or convenient to use a host processor that has the same instruction set as the target processor, a *cross-profiling* technique was developed [12]. In this approach, compilers for a common source language for both the target and host machines are required. The target compiler can be a cross-compiler (i.e., it can run on the host machine) or a native compiler (running on the target machine). We call the assembly language outputs of the target and host compilers the target assembly language representation (TALR) and host assembly language representation (HALR), respectively.

Cross-profiling assumes that there is a close correspondence between basic blocks at the source code level and basic blocks at the assembly language level. Although this correspondence is rarely exact in practice, it is usually close enough to establish a nearly one-to-one correspondence between blocks of the different assembly language representations. That is, each basic block in the source code has a corresponding block in both the TALR and the HALR. This correspondence between assembly language representations is then used to incorporate information from a timing analysis of the TALR into the HALR by assigning the cycle counts derived from the basic blocks of the TALR to the corresponding basic blocks in the HALR.

Differences in basic block structures between the source program and the assembly language representations are frequently due to redundant basic blocks in the assembly code that do not correspond to any basic block in the source program and are therefore unlabeled. A similar type of mismatch between the basic blocks of the TALR and the HALR is caused by one compiler producing unnecessary basic blocks that are not produced by the other compiler. An example would be a basic block B that can be entered only as a result of a jump from the end of another basic block A. Consequently, A and B are always executed together and in sequence. These differences can often be eliminated by performing a basic block reduction on both the TALR and HALR that coalesces certain blocks in the graphs without changing their control flow. Note that basic block reduction is also of some value in standalone profiling, since it removes the dynamic time accounting overhead from all but one of the basic blocks that are combined. Experiments using a SUN Workstation as the host processor and cross-profilers for two different target processors and three different C compilers compared the total accumulated cycle counts of cross-profiled sequential programs to the cycle counts for native profiling of the same programs with the same data. These experiments showed errors due to mismatched basic blocks after block reduction that ranged between 0.044% and 0.85% [13].

The cross-profiling technique is illustrated in Fig. 2. The first step is to detect the basic blocks in the HLL source program and mark each with a unique label. Then the source program is compiled twice, once into the TALR and once into the HALR. The basic block labels in the source program are passed through to the assembly language code unchanged by the compiler. The correspondence between basic blocks in the two assembly language representations will be established later by matching identical labels in the two representations.

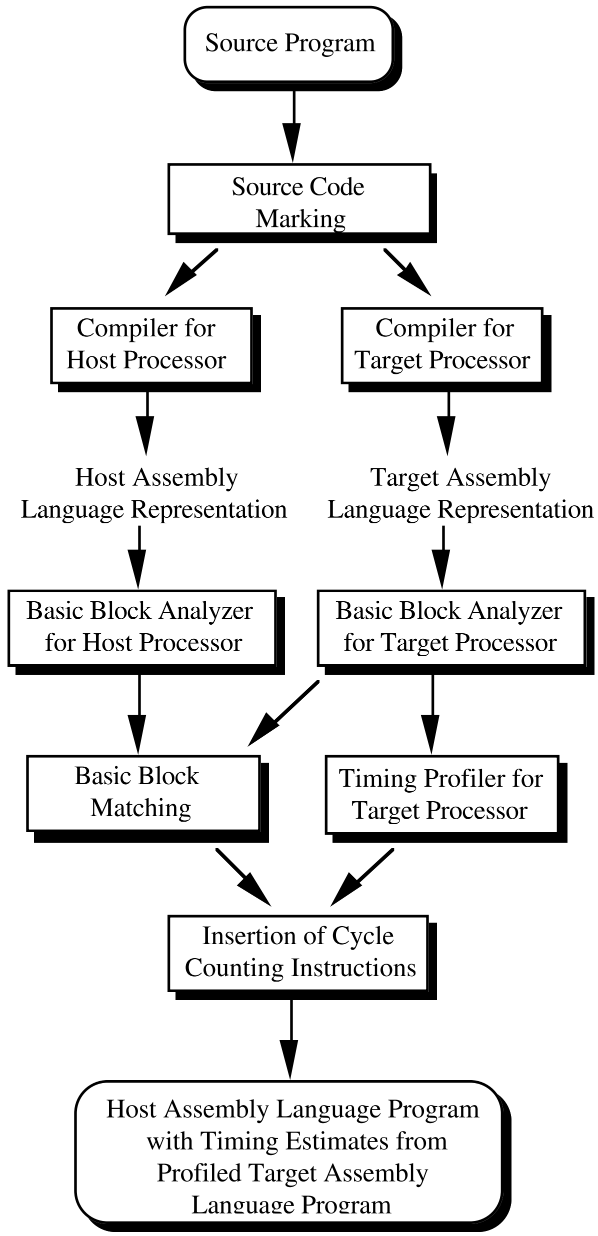

Fig. 2. Cross-Profiling.

Next, the basic block structures of both the TALR and HALR are identified and reduced by the basic block analyzers, and a timing analysis is performed on the TALR to produce estimates of the number of cycles needed to execute each basic block. Timing analysis is not performed on the HALR since it will use the timing estimates from the TALR when the simulation is performed. The basic blocks of the TALR and HALR are then paired using the basic block labels. Finally, the HALR is instrumented with cycle counting instructions in the same way as with the standalone profiler. However, in cross-profiling, the incremental values used by these counting instructions are those produced by the timing analyses of the TALR instead of the HALR. When the simulation model is executed, the timing estimates provided by the profiled program reflect the program's execution time on the target processor rather than the simulation host.

One of the first simulators to utilize execution-driven simulation was Fujimoto's Simon simulation system [9]. It uses a form of execution-driven simulation called *native mode execution* to allow efficient evaluation of the performance of parallel programs executing on multiprocessors. Fujimoto also developed a technique for simulating systems where the host and target processors have different instruction sets [14]. Yet another approach for dealing with heterogeneous host and target processor instruction sets, similar to the one utilized in the RPPT, is described in [15].

The basic concept inherent in execution-driven simulation was also used by Williams and Bobrowicz [11] to simulate an N-processor Cray X-MP with a simulator running on only one of the processors of a Cray X-MP/24. They used measured user code execution times to drive the simulation. An execution-driven approach was also proposed by [7] as a means of dynamically generating and interleaving multiprocessor traces. They discuss several methods for trace interleaving in uniprocessor simulations of multiprocessor systems and refer to their approach to execution-driven simulation as *on-the-fly tracing*.

The computer architect's workbench [10] uses execution-driven simulation techniques to study the relative performances of a variety of uniprocessors. Here the program source language is compiled to an intermediate U-code, which is analyzed for basic blocks and timing estimates based on a set of assumptions about the target processor architecture, and then the u-code is translated into assembly language code for the simulation host processor. One important feature of the Workbench is that it offers the user a choice of application source languages - C, Pascal, or Fortran.

SIMPLE/CARE, developed at the Knowledge Systems Laboratory at Stanford, is an event-based simulation system for studying the performance of multiprocessors or multicomputers with 100-1000 processors, with a primary focus on parallel expert systems [16]. LAMINA is an applications programming interface for SIMPLE/CARE, built upon Zetalisp [17]. Execution objects in LAMINA applications can either be assigned fixed execution times or use the simulation host clock to dynamically determine them.

3. The Rice Parallel Processing Testbed

To perform a simulation with the RPPT, a user must supply three components: a *program module*, an *architecture model*, and a *process mapping* used by the RPPT to assign the processes of the parallel program to the physical processors of the architecture. Once these are specified, the RPPT assembles an *RPPT simulation module* that can be executed to perform the actual simulation and obtain performance and timing information for the specified program executing on the specified architecture. When this module is executed, it produces two types of output data. First, it generates trace data that characterizes the history of the program's execution and is used primarily for debugging. Second, it produces information on event times and resource utilizations which is used to generate performance metrics of interest. The types of trace and timing data to be collected are specified by the user as part of the program and the architecture models.

Both program modules and architecture models can be developed and placed in libraries for future use. The RPPT user can then assemble a simulation module by selecting program modules and architecture models from these libraries. While it may be that some programs will not be appropriate for some architectures, both can usually be written in a general way, so that they are relatively independent of each other. We have found this to be true for programs and architectures that use message passing for communication between processors.

3.1. RPPT Architecture Models

A parallel architecture model for the RPPT consists of *processor modules* (usually with local memory), *global memory modules*, and an *interconnection structure* that provides communication paths between processor and memory modules. Since the RPPT does not simulate the internal behavior of processors, the major component that must be specified to define an architecture model is the interconnection structure. In fact, an architecture model for the RPPT is defined by specifying the number of processor and memory modules and writing a simulation program named *UserSend* that specifies the behavior of the interconnection structure and how the processor and memory modules are interconnected. (The type of processor, which also can be thought of as part of the architecture specification, is used to select an appropriate profiler and hence becomes part of the program module.)

UserSend is written in a simulation language called *CSIM* [18]. This is a general purpose discrete-event simulation language that uses a process-interaction approach and is implemented as an extension of Concurrent C (see the following section) with a set of procedures for event queue manipulation and statistics collection. It is primarily a library of subroutines that implement various simulation functions for the processes in the simulation model. For instance, a process may suspend its own execution until a specific simulation time, it may schedule another process to be activated at some future simulation time, or it may delay its further execution until a particular condition becomes true. The ability to delay on a condition value is one of CSIM's most powerful features. Conditions are C language expressions involving state variables (defined by the user) and variables describing the states of CSIM objects such as semaphores and queues. CSIM also provides statistics collection mechanisms and a simulation trace facility.

UserSend models the overhead of accessing the interconnection network, the effects of contention due to multiple simultaneous users of the network, and the transmission time for the message. UserSend can be simple, as in the case of a shared bus communications channel that is modeled as a FIFO queue with service time proportional to the message length, or it may be quite complex, for example to describe all of the activity in a multistage interconnection network. It is in writing UserSend that we must be concerned about the details of the implementation of the parallel system.

UserSend can be written so that many of the parameters that characterize the behavior and structure of the interconnection structure (e.g., bus or channel bandwidth), can be specified as run time parameters. These parameters can then be specified to the RPPT just before it assembles a simulation module, thereby allowing the user to collect data over a range of program parameters without remaking the architecture module each time a parameter is changed.

Processor modules are not constructed by writing simulation code that emulates the internal operation of a processor. Instead, a timing profiler that knows about the instruction times of the processor modifies the program to be simulated by inserting cycle counting instructions as described in Section 2. The effect of this is that the processor module is embedded in the program module described in the next section, and an RPPT user only needs to specify the number of processors in the architecture module.

Global shared memory modules are difficult to simulate efficiently, since access times can not be predicted accurately by the profiler due to contention in the interconnection module and at the memory modules. Consequently, UserSend and a detailed memory simulator must be called for each access to global memory, and this can happen several times within a basic block. This can significantly reduce the efficiency gains derived from the execution-driven approach. As a result, we have concentrated initially on message-based systems that do not have global memory. A discussion of some of the simulation problems due to global memory and some proposed solutions can be found in reference [19].

3.2. RPPT Program Modules and Process Mappings

Parallel programs for the RPPT are usually written in a language called *Concurrent C (CC)* [20] (Presto [21] and a parallel version of Fortran based on a message-passing paradigm have also been used). Concurrent C is an extension of the C programming language that provides primitives for process control and communication. It is implemented as a library of subroutines and predefined data types that extend the standard C language to provide a pseudo-concurrent execution environment. Processes defined in Concurrent C all run in the same address space, and hence context switches incur relatively little overhead. The user can create, activate, suspend, fork, and join processes. A process can interact with other processes by sending and receiving messages (both asynchronous and synchronous message passing are supported) or through the use of semaphores.

When processes are created or global data declared in a Concurrent C program, they are assigned to logical nodes rather than to specific processors or memory modules, allowing the user to write a concurrent program without prior knowledge of some of the specific details of an architecture on which the program will be executed. This can greatly simplify the task of evaluating the performance of a single parallel program on different architectures. The user specifies a mapping of logical nodes to physical nodes (processors and memory modules) as one of the basic components of an RPPT simulation module. If two processes are assigned to the same logical node, then they are guaranteed to be execute on the same physical node. However, if they are assigned to different logical nodes, they may still be assigned to the same physical node by the logical-to-physical node mapping.

Concurrent C programs can also be written to use run time parameters to simplify the simulation of a program for a range of data sizes or program options. As with architecture models, these parameters are then used by the RPPT to assemble a specific simulation module.

Fig. 3 shows the procedure for converting a Concurrent C program into an RPPT program module. It uses the RPPT components ASIMP and TPROF or XPROF that serve as translators for converting programs from one form to another and inserting information needed to simulate the execution of the original program. The operation of these translators is summarized below.

The Architecture Simulation Preprocessor (ASIMP):

ASIMP is a source code translator that modifies Concurrent C programs by inserting primitives used to simulate the architecture delays due to communication and synchronization between processors. It essentially converts the parallel program into a simulation program that knows about the architecture that is being simulated. It does this by scanning the program to find all points at which one process interacts with another or at which a process accesses a global memory module. At each such point, it inserts code that will use the logical-to-physical node mapping at run time to decide whether or not the operation requires data communication (including access to shared variables for synchronization) between processors. If it does, the inserted code will call the procedure UserSend to simulate the data movement. Otherwise, it will complete the operation by calling on the appropriate Concurrent C routine.

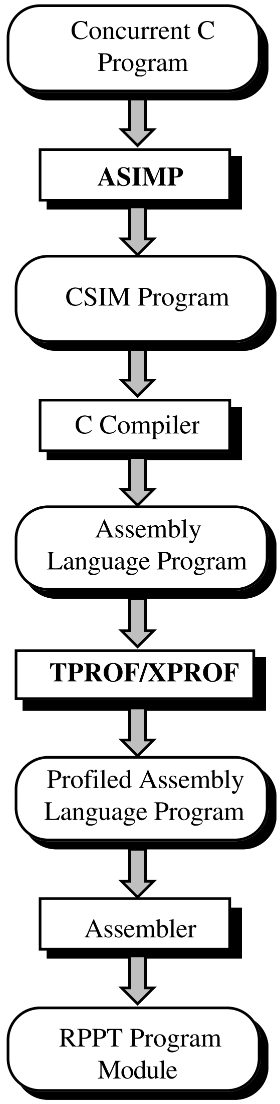

Fig. 3. Conversion of a Concurrent C Program into a Program Module.

The Timing Profilers (TPROF and XPROF):

TPROF is a standalone profiler and XPROF is a cross-profiler of the type described in Section 2 above. While the combination of ASIMP and UserSend accurately accounts for simulation time and resource contention in interprocessor communication, TPROF accounts for elapsed time due to instruction execution within the processors. The modified source language program produced by ASIMP is compiled by the C compiler, and the timing profiler analyzes the assembly language output of the compiler and inserts the counting instructions used to accumulate the execution time of basic blocks. These profilers are based on the bb profiler developed at Bell Labs [22].

A separate cross-profiler is needed for each target instruction set that is different from that of the simulation host. If we are to use the processor of the RPPT host as a target processor, we will also need a standalone profiler for it. If the RPPT is ported to a new host system, then we need a new profiler for the host, but we do not need new target code profilers.

In some cases, it is desirable to allow the user to directly specify the execution time for a procedure. For example, this may be the only way for a user to account for the amount of simulation time that various operating system calls require when their source code is unavailable. To implement this, the profilers have been designed so that selected procedures can be left unprofiled. For these, the user must specify a function that can be used to calculate the time taken to execute the procedure each time it is executed.

3.3. RPPT Simulation Modules

The construction of an RPPT simulation module is illustrated by Fig. 4. This is done by another basic RPPT component, the *Simulation Tool Interface (SIMTOOL).* This is the main user interface for the RPPT. It is a menu-driven, window-oriented program that runs on a Sun workstation and simplifies the task of combining a program module, an architecture model, and a node mapping into a simulation module. It has been developed to provide the user with a structured guide through the process of constructing the simulation module.

SIMTOOL first prompts the user to select a program module and an architecture model from libraries. To simulate an architecture or program that is not available in these libraries, the user must construct the modules as described above and place them in the libraries. SIMTOOL also prompts the user for any parameters needed for the architecture model or program module. Next, SIMTOOL asks for a specification of the node mapping. This can be given by specifying a file that contains the mapping or by entering a formula that can be used to translate a logical node number to a physical node number. It is also possible to specify whether or not trace data will be generated and where it is to be directed. It can either be collected in a file for later analysis or piped to a process-level debugger for run-time display of the process activity.

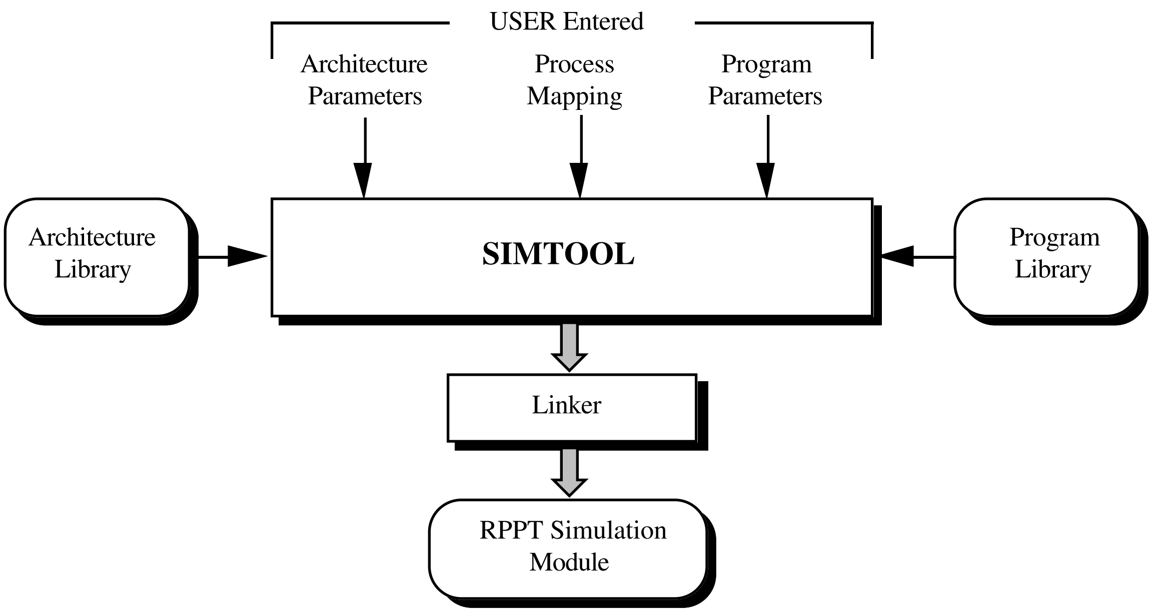

Fig. 4. Construction of a Simulation Module.

Once all the inputs to SIMTOOL have been entered, SIMTOOL assembles them into the simulation module and passes them to the linker which links in unprofiled procedures such as those that implement the Concurrent C and CSIM primitives. The final result is a module that can be executed to perform the simulation and produce the trace and timing data.

3.4. Simulation Tracing and Debugging

Debugging parallel programs is considerably more difficult than debugging serial programs. Multiple threads of execution and synchronization between them add to the complications one normally finds in serial programs. Traditional symbolic debuggers for serial programs typically allow the user to step through the execution of a program and examine or modify the state of the program at any point. A straightforward application of this strategy to the debugging of parallel programs results in flooding the user with a large amount of information. Moreover, in a true parallel system it is often not possible to take a snapshot of the state of all the processes simultaneously, as this would require a global synchronizing mechanism.

The problems due to multiple threads of execution imply that an effective debugger must be able to help the user keep track of multiple processes and present information about each in a usable manner. The user should, at a glance, be able to perceive the state of all, or at least the most important, processes and should be able to specify which information about these processes is considered relevant. For instance, experience suggests that incorrect synchronization between processes is a common error in parallel programs. Thus a parallel debugger should allow the programmer to select information about the synchronization points in a process' lifetime for display.

TRAPP is a trace/debug tool for RPPT programs. It takes trace information generated by a simulation and displays the traces for different processes in separate windows on a Sun workstation. Trace information consists of significant events in the lifetime of a process such as process creation and termination, synchronization with other processes by means of semaphores, and message passing. Each event displayed in a process window is also time- stamped with its simulation time of occurrence.

Because the generation of TRAPP input data during a simulation is non-intrusive, this approach offers significant advantages over techniques for generating similar data on real machines. The addition of the instructions necessary to produce and collect trace information does not change the behavior of the simulated system by affecting process execution times, as can occur in a real system. TRAPP can be used to monitor both program and architecture activity, since the program modules and architecture models can include statements for generating various types of trace information. Also, simulations using the RPPT and TRAPP are deterministic and hence repeatable. If a parallel program does not execute correctly and the information that it produces is not sufficient to isolate the problem, the simulation can be repeated to generate additional trace information and we can be sure that the behavior of the simulated system will not change. This is in contrast to debugging real parallel systems, where the insertion of additional code to attempt to find a bug may in fact mask the original bug or create (or unmask) other problems.

4. RPPT Performance

If the RPPT is to be an effective tool in studying parallel systems, it must produce accurate performance measurements and it must do so with "reasonable" efficiency. Consequently, a major, and in some ways the most difficult, part of the RPPT development has been the validation of simulator performance predictions against measurements obtained on real systems. This section describes some of the validation experiments that were conducted and presents results on both the accuracy and efficiency of the simulations.

The choices of real systems to be used in the validation studies were limited to the parallel systems that were available to us. These were a 16-node Intel iPSC hypercube and the Loosely-Coupled Multiprocessor (LCMP), a set of Sun workstations interconnected by an Ethernet and running the distributed V system kernel [23]. Neither system is ideal for this purpose. A major problem that they have in common is the lack of a high-resolution clock. The iPSC was also difficult to work with because the interprocessor communication mechanism caused some interference with (i.e., delayed) the user processes executing on the processor nodes. This was especially true for multi-hop interprocessor communications; that is, those communications in which the source and destination nodes are not directly connected. The workstations used in the V system studies were connected to an Ethernet that also served a large number of other systems, and thus it was never possible to be completely sure that background communication traffic was not biasing the validation results.

A large number of parallel programs have been developed for the RPPT, and we chose both numeric and non-numeric programs to use in the validation effort. In each case, the algorithms were executed on the real system for several input data set sizes. Usually we used the largest number of processors available on these systems, but in some cases smaller numbers of processors were used as well. The algorithm executions were also simulated on the RPPT, running on a Sun 3 (68020-based) workstation. For a given architecture, the simulation model and the timing profiler were fixed. Both studies required the use of cross-profiling. The processors in the iPSC are Intel 80286/80287 pairs. The compiler for the 80286 actually runs on the iPSC, and the assembly language output, with inserted native profiler marks and cycle counts, was transferred from the hypercube to the Sun workstation for basic block matching. The processors running the V system were 68020's, identical to the host processor, but the V system C compiler and Unix C compiler are not identical, again requiring that a cross-profiling step be performed.

4.1. Hypercube Validation

The Intel iPSC hypercube is a distributed memory multiprocessor with 16 nodes, each an 80286/80287 pair with local memory. Each node is physically connected to four other nodes. Nodes communicate via message passing. Messages longer than 1 Kbyte must be broken into packets of length 1 Kbyte or less for transmission.

When a node sends a message to another node that is not directly connected to the sender, the packets must be relayed by each node along the multi-hop path from sender to receiver. At each hop, the packet must be completely received by the intermediate node before it can be forwarded on the next hop. The act of relaying a packet requires that the intermediate processor interrupt the currently executing process (if there is one) for the time required to examine the received packet and arrange to forward it.

The CSIM architecture model used in the iPSC studies uses the same fixed message routing scheme used in the iPSC. It also correctly breaks each message into packets (if necessary). The time to transmit a message includes the software overhead to break the message into packets at the sender and reassemble it at the receiver, the time to transmit packets over the communication links along their paths, and the time to process each packet at all intermediate nodes along the path. However, the model does not correctly take into account the delay experienced by a user process which is interrupted because of a packet being forwarded through the node on which that process is executed. Since this overhead is apparently roughly constant for any packet length, the performance predictions for algorithms which involve the transmission of a relatively large number of multi-hop packets will be most strongly affected by this modeling inaccuracy.

A number of algorithms have been implemented on the iPSC and simulated on the RPPT with the iPSC model. In this paper we present validation results for five representative parallel programs executed on the iPSC. These are: (1) eigenvalue/eigenvector computation; (2) LU decomposition; (3) Radix-2 FFT (fast Fourier transform); (4) mergesort; and (5) Global Distribution Local Sort (GDLS). The first three algorithms are numerically intensive, while the latter two include little arithmetic other than simple integer comparisons. Details of these algorithms can be found in the following theses [12, 24, 25].

The eigenvalue/eigenvector computation (*Eigen*) is characterized by single-packet, single-hop communication patterns, although occasionally multi-hop transmissions are used. The program is broken into three components. The first, a tridiagonal reduction, splits the data into p (= the number of processors) parts to be processed. For small data sizes, the communication aspects dominate, but as the data sizes get larger, each processor does relatively more computation per unit of communication, and eventually the computation dominates. The second component is an eigenvalue estimation phase in which communication occurs infrequently, and the third component is a QR iteration phase which is entirely computation intensive. The last two components are iterative, with run times determined by the data and the convergence criteria. The tridiagonal reduction takes time proportional to the size of the data.

The LU decomposition algorithm also is largely characterized by single-packet, single-hop message transmissions with occasional multi-hop communications. It actually consists of two parts: the LU decomposition itself, which behaves somewhat like the tridiagonal reduction described above, and a back and forward substitution step which is communication intensive.

The FFT algorithm is highly compute-bound, and all communication is nearest neighbor (i.e., between directly connected nodes).

The parallel mergesort algorithm involves first partitioning the data into p sets, which are sorted sequentially on the p processors. It then goes through $\log p$ iterations in which the data are merged in successively larger sets. Each iteration begins with a communication phase in which single packet messages are sent over paths that grow longer with successive iterations. This phase is followed by a series of message transfers involving large amounts of data and a fairly random number of hops.

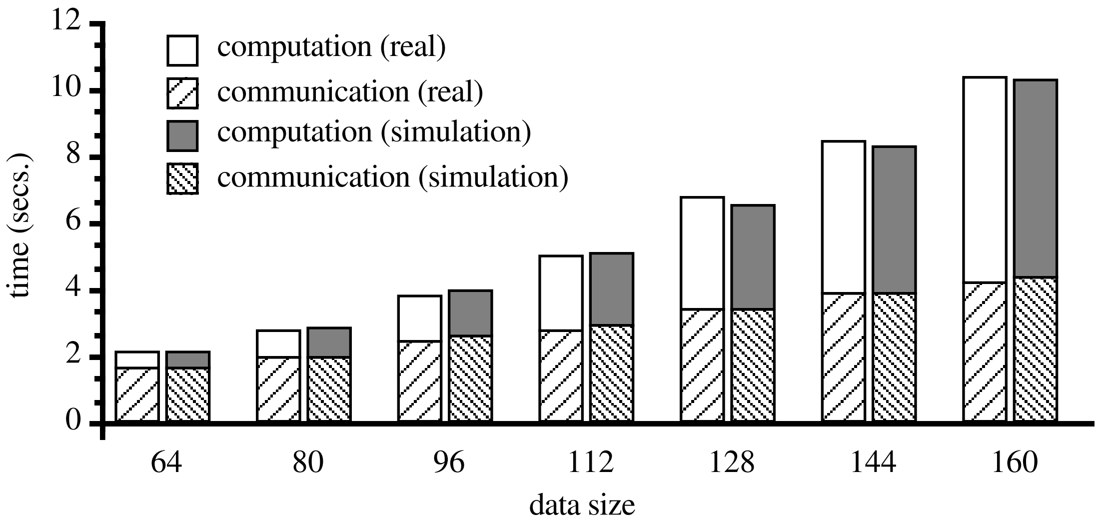

Fig. 5. Validation - Eigenvector/Eigenvalue (iPSC; 16 Processors).

Fig. 5 shows the results for the Eigen algorithm, with the input matrix size varying from 80 x 80 to 160 x 160. The graph breaks the total execution time down into computation time and communication time for both real system program execution and simulated execution. As expected, the computation time dominates over the entire range. This is in contrast to the results obtained for the LU decomposition (Fig. 6), where the communication time dominates for small data sizes (beginning with 64 x 64 arrays) and is still a significant factor at 160 x 160, although the computation component is increasing more rapidly than the communication. The FFT algorithm (Fig. 7) is so completely dominated by computation time that the accuracy of the RPPT simulations is determined almost entirely by the timing profiler accuracy.

The simulation accuracy for these floating point-intensive algorithms is shown in Figures 8 and 9. The relative error is the difference between the execution times as predicted by the simulator and measured on the iPSC, normalized by the measured execution time. The relative error is within -7% to +3% for all three algorithms, and the Eigen error is virtually constant at between -4% and -2%.

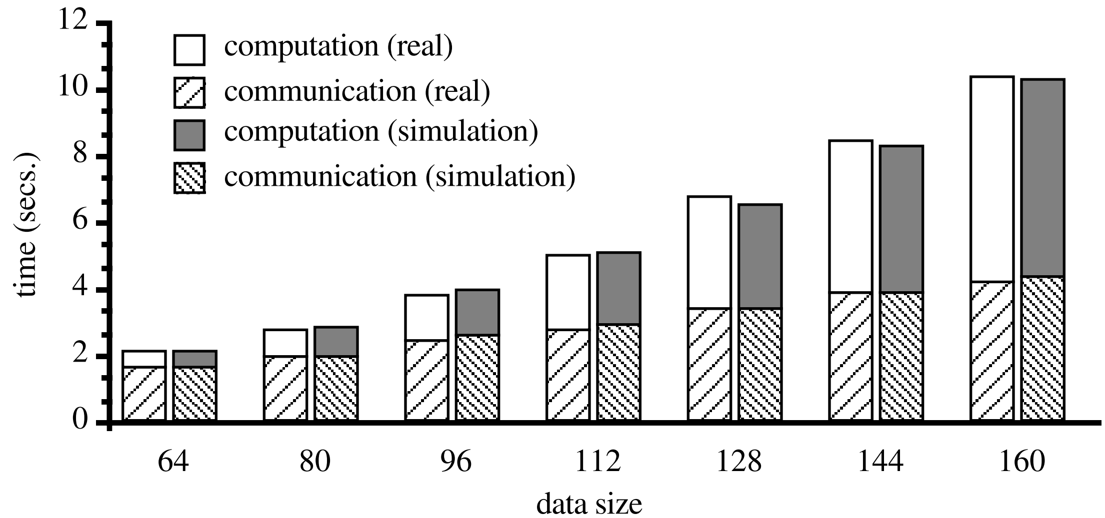

Fig. 6. Validation - LU Decomposition (iPSC; 16 Processors).

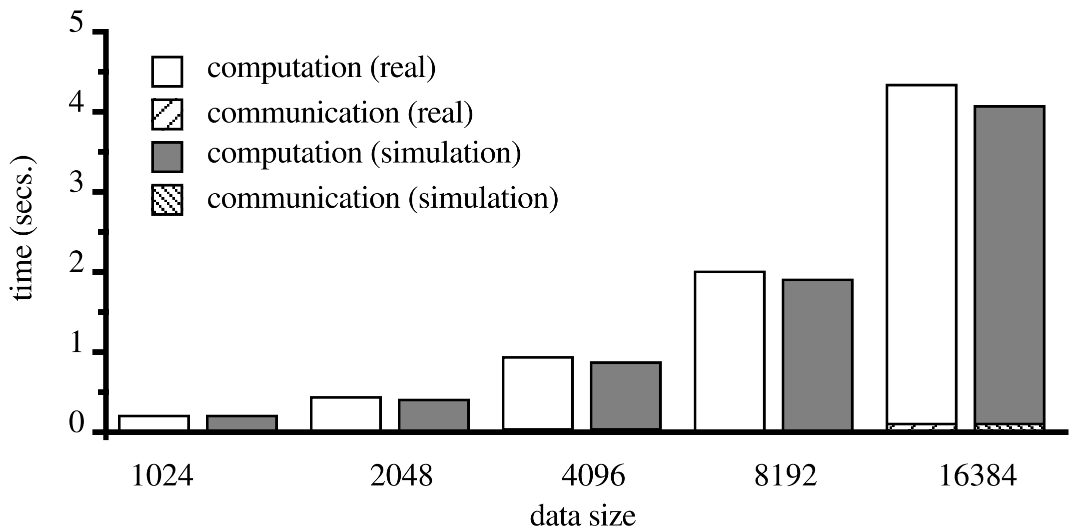

Fig. 7. Validation - FFT (iPSC; 16 Processors).

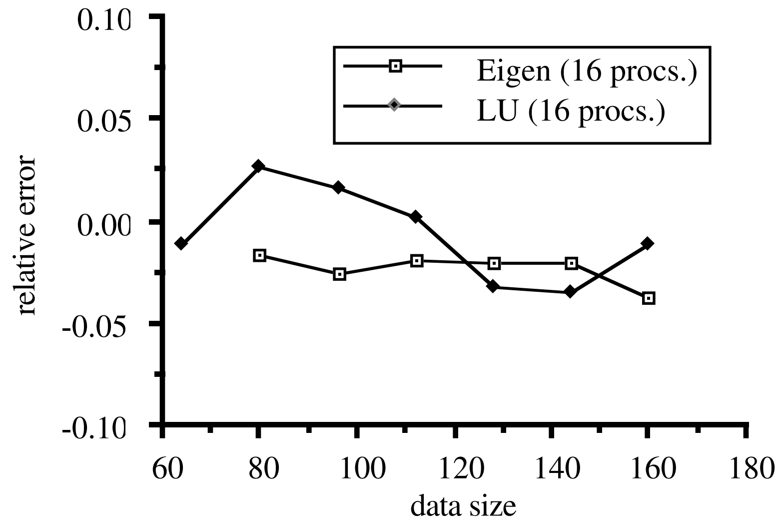

Fig. 8. Relative Errors - Numeric Algorithms (iPSC).

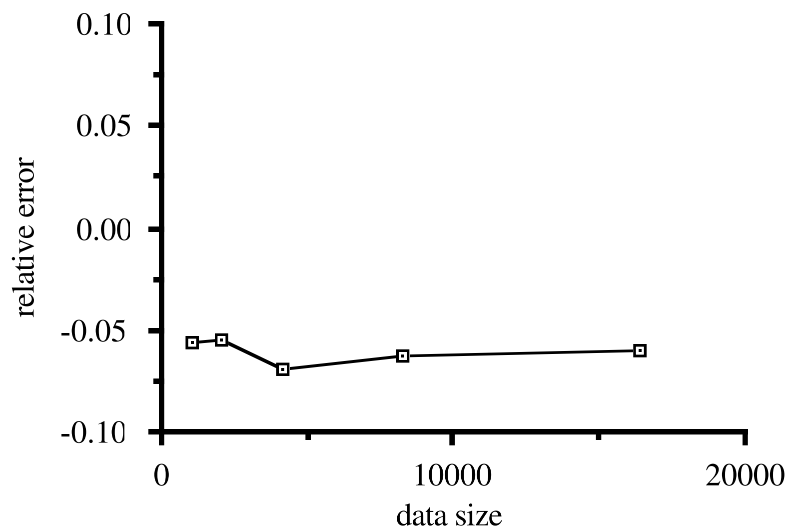

Fig. 9. Relative Errors - FFT (iPSC; 16 Processors).

The results for the non-numeric algorithms are shown in Figures 10-12. The first two figures compare the measured and simulated execution times for the two sorting algorithms implemented on the iPSC. For the parallel mergesort algorithm, communication time dominates for all problem sizes, and the inaccuracies in the iPSC model have a greater impact on the simulator predictions, as seen in Fig. 12 where the relative error is approximately -8% to -10%. Although the results look somewhat better for the GDLS, especially for large data sizes where there is almost no error, it must be remembered that there may well be some cancellation between architecture modelling and profiling errors.

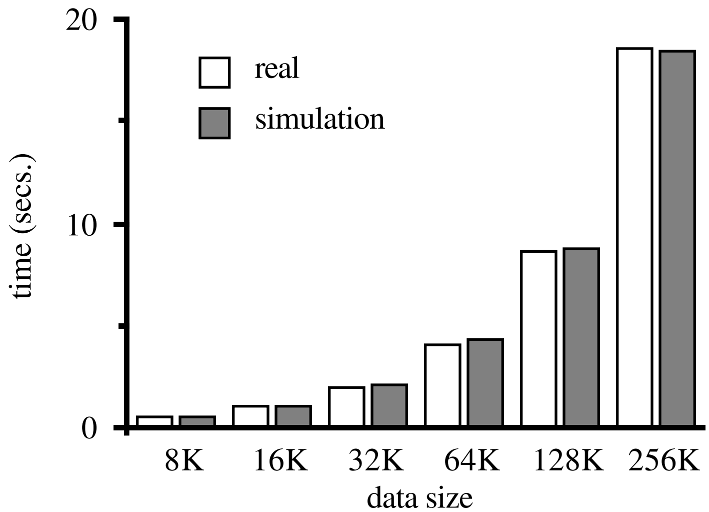

Fig. 10. Validation - GDLS (iPSC; 8 Processors).

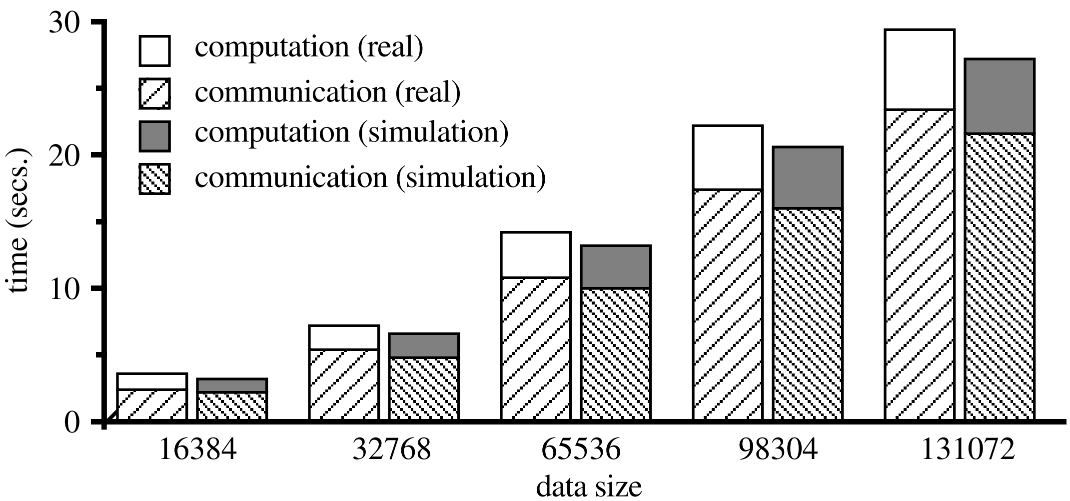

Fig. 11. Validation - Merge Sort (iPSC; 16 Processors).

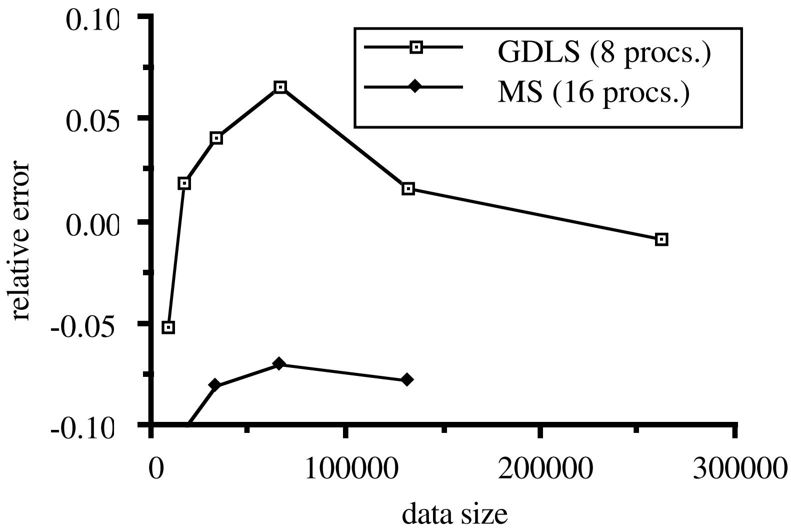

Fig. 12. Relative Errors - Sorting Algorithms (iPSC).

4.2. LCMP Validation

The LCMP is difficult to use for validation purposes. Since the workstations that make up the LCMP are connected via a public Ethernet, communication times are sometimes greatly affected by non-LCMP traffic. The architecture model for the LCMP is that of a shared bus with collision detection and backoff, but it does not take into account the possibility of collisions with non-LCMP traffic. To minimize the modeling discrepancies as much as possible, the validation experiments were performed when the Ethernet was expected to be lightly loaded. Each experiment was run several times, and the minimum execution time run was chosen as the most likely to match the results from the RPPT model.

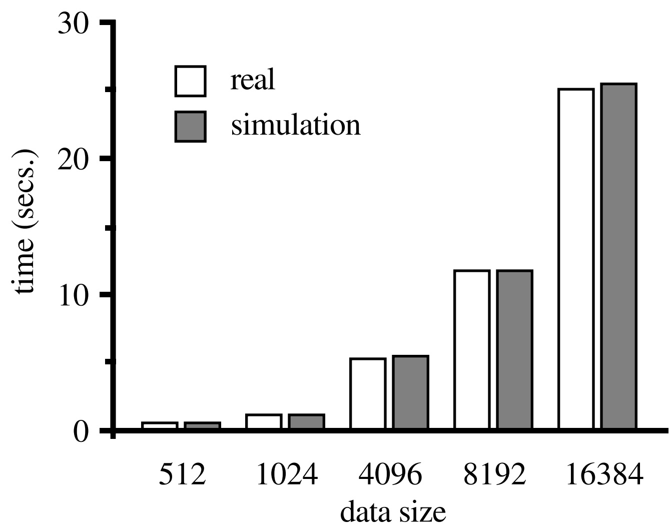

Fig. 13. Validation - FFT (LCMP; 4 Processors).

In addition to the Radix-2 FFT and GDLS algorithms described above, we used a fast Hadamard transform (FHT) [26]. The FHT utilizes floating-point adds extensively and has relatively little communication, like the FFT. (The LCMP is best suited for algorithms with this very coarse-grain type of parallelism.)

The results for the FFT and FHT are very similar (Figures 13 and 14). Again, these are primarily a reflection of the accuracy of the timing profiler (and the cross-profiling technique) rather than the architecture model. This is clearly shown in Fig. 15, where the relative accuracy stays within about -5% to +2%, with one exception. The somewhat inconsistent point at a data set size of 128K for the FHT may be largely due to background traffic on the Ethernet. The relative errors tend to be much larger for the GDLS (Figures 16 and 17), although for larger data sizes the error seems to be approaching -8%.

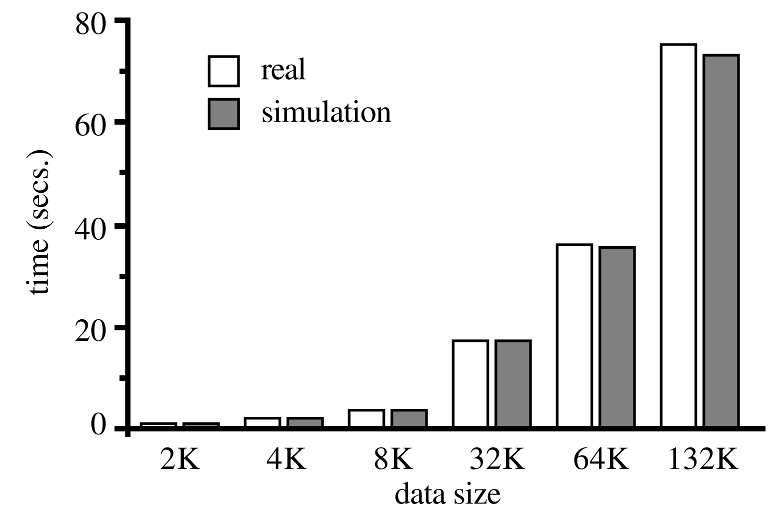

Fig. 14. Validation - FHT (LCMP; 4 Processors).

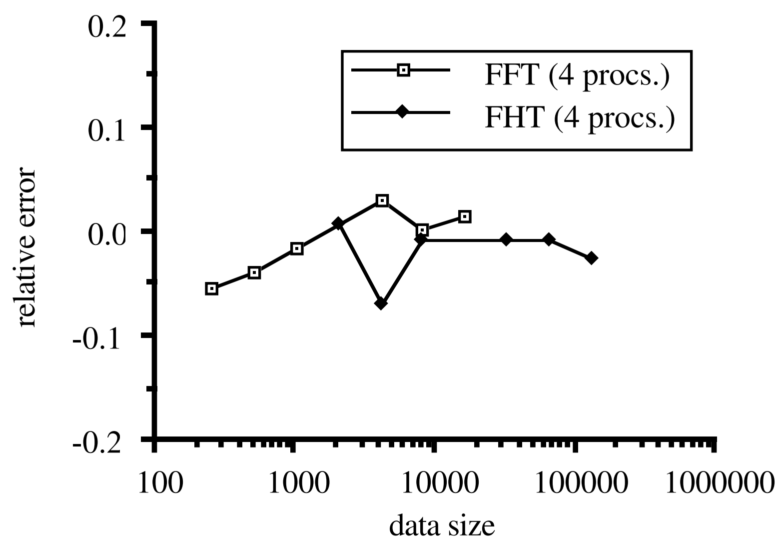

Fig. 15. Relative Errors - Transforms (LCMP).

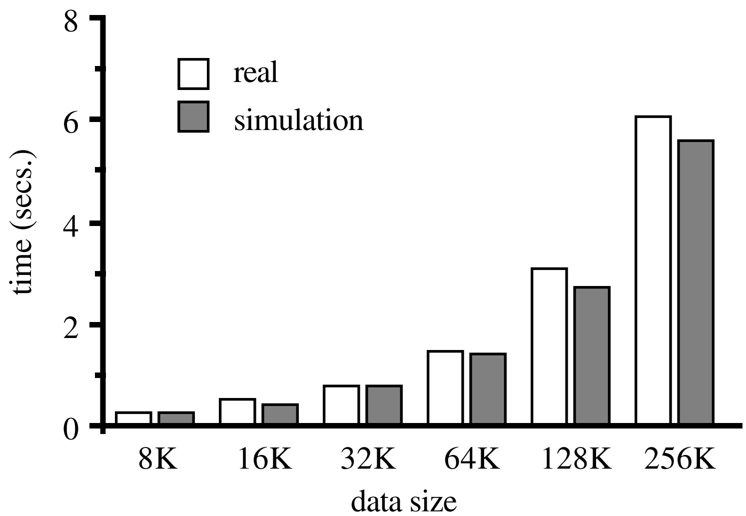

Fig. 16. Validation - GDLS (LCMP; 5 Processors).

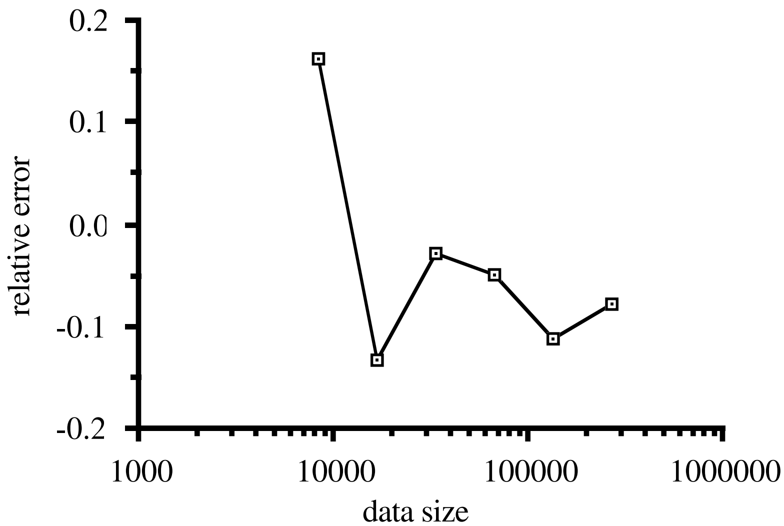

Fig. 17. Relative Errors - GDLS (LCMP; 5 Processors).

4.3. RPPT Efficiency

An RPPT simulation introduces two types of overhead in the execution of a parallel program. The first is the profiling overhead - the instructions inserted by the timing profiler that accumulate cycle counts between process interaction points. For simulating distributed memory multiprocessors with no caches, this overhead is generally fairly small and does not vary a great deal from one program to another. It is a function of the average size of a basic block in the program and is less for programs which frequently use instructions with larger execution times (for example, floating point).

The second type of overhead is due to the architecture simulation. At each process interaction point, such as a message send or receive, ASIMP inserts a call to a routine that determines whether or not the interaction is with a remote process, and if so invokes the architecture model to account for the time to accomplish remote interaction. This type of overhead is highly variable and is influenced heavily by the structure of the program itself, as well as the complexity of the architecture model. Programs in which processes communicate with one another frequently incur significantly higher overheads as RPPT applications than do programs in which processes are more computation-intensive. The overhead does not depend strongly on the amount of data transferred.

The RPPT efficiency is measured by the slowdown ratio, which is determined as the ratio of the time required to execute the RPPT simulation of a program's execution to the time required to execute that parallel program as a standalone CC application. CC provides a pseudo-parallel runtime environment in which the unprofiled user's program is executed. Message passing and other parallel programming primitives incur very little overhead since the pseudo-parallel processes share the same address space and no architecture simulation overhead is incurred.

Table 1 presents slowdown ratios for each of the algorithms described above. For brevity, we have included only a single data point for each algorithm, based on the maximum number of processors used for that algorithm and a large data size. Slowdown ratios for smaller data sizes tend to be larger, since there is some overhead involved in starting up an RPPT simulation that is independent of the length of the simulation.

The slowdown ratios are in the range 1.22 to 2.209, with one exception. The result for the LU decomposition algorithm (a slowdown ratio of more than 14) is due to the poor structure of the algorithm, which sends large numbers of relatively short messages between processors, both in the decomposition phase and especially in the subsequent back and forward substitution phase. Note that the parallel mergesort algorithm only has a slowdown ratio of 2.2, even though Fig. 11 shows that it has an even worse computation-to-communication ratio than the LU decomposition algorithm. This is because the mergesort uses relatively few but long message transfers, which account for a large amount of simulated time but actually take little time to simulate.

	Algorithm	Number of processors	Size of problem	Slowdown
LCMP	Radix-2 FFT	4	16,384	1.41
	FHT	4	131,072	1.22
	GDLS	5	256,000	1.9
iPSC	Radix-2 FFT	16	16,384	1.205
	Eigen	16	96 x 96	1.607
	LU Decomposition	16	96 x 96	14.31
	Mergesort	16	65,536	2.209
	GDLS	8	131,072	1.774

Table 1. RPPT Slowdown Results.

5. Conclusions

The main goal of the research reported here was to demonstrate the effectiveness of a relatively new simulation technique, called execution-driven simulation, for simulating parallel computer systems. This technique is characterized by higher accuracy than distribution-driven simulation and better efficiency than instruction-level simulation. The effectiveness of the technique has been demonstrated by implementing a sophisticated and powerful simulation system called the Rice Parallel Processing Testbed (RPPT).

The RPPT not only confirms the claimed advantages for execution-driven simulation, but is a useful tool for investigating the relationships between the structures of parallel algorithms and different parallel computer organizations. It can be used to study the effect on system performance of both major and minor changes in the structure of a parallel architecture. For a given architecture, it can be used to evaluate the relative performance of different parallel algorithms as well as the effect of minor changes in a given algorithm. As a result, it provides a powerful quantitative tool for matching parallel algorithms and architectures to optimize the total system performance.

The experimental results presented in Section 4 are a major contribution of the paper. They are the result of extensive experiments designed to measure both the accuracy and efficiency of the execution-driven approach. Variations of two different message-based architectures, a hypercube and a loosely coupled, distributed mutiprocessor system were investigated. Four numeric and two non-numeric parallel programs were used to drive the simulations. The results verify the advantages claimed for the execution-driven technique.

The accuracies of the RPPT predictions were determined by comparing measurements of the time to execute the parallel programs on real parallel systems to predictions of their execution times produced by the RPPT. These comparisons demonstrated differences of less than 10% in all cases and less than or equal to 5% in most cases where the size of the data manipulated by the programs was large enough to mask startup effects.

The efficiency of the RPPT was evaluated by measuring the overhead added to a program's execution time by the simulation code added by the preprocessors. This was represented by the ratio of the time required to execute a program with inserted simulation statements to the time required to execute it without them. This ratio, called the slowdown ratio, was shown to range from 1.2 to 15 for the architectures and programs that were used. This compares favorably with the slowdown of instruction-driven simulations that can range into the hundreds.

Execution-driven simulation is limited in accuracy primarily by the degree of detail incorporated in the architecture model and the ability of the profiler to determine basic block timings precisely. The latter is especially important when dealing with cache memories, where there is a range of tradeoffs to be made between efficiency and accuracy. Also, simulation of a parallel processor on a sequential processor means that execution of the parallel program driving the simulation is necessarily serialized. Although this does not affect the correctness of the results, it does affect the time to execute the simulation. Perhaps a more serious problem is that since the real program is being executed, enough memory is needed to hold all of the program text and data that would be used on the parallel processor. This may limit the size of a simulation experiment, even in a virtual memory environment, because of constraints on the amount of physical memory and/or swap space.

References

[1] K. So, F. Darema-Rogers, D. A. George, V. A. Norton and G. F. Pfister, "PSIMUL: A System for Parallel Simulation of the Execution of Parallel Programs," Research Report RC11674, IBM T. J. Watson Research Center, Jan. 1986

[2] T. Axelrod, P. Dubois and P. Eltgroth, "A Simulator for MIMD Performance Prediction: Application to the S-1 MkIIa Multiprocessor," *Parallel Computing*, Vol. 1, pp. 237-274, 1984.

[3] J. M. Butler and A. Y. Oruc, "A Facility for Simulating Multiprocessors," *IEEE Micro*, Vol. 6, No. 5, pp. 32-44, Oct. 1986.

[4] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Computer Systems*, Vol. 4, No. 4, pp. 273-298, 1986.

[5] Y.-B. Lin, J.-L. Baer and E. D. Lazowska, "Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols," in *Proc. 1989 SCS Multiconference on Distributed Simulation*, 1989.

[6] S. J. Eggers, E. D. Lazowska and Y.-B. Lin, "Techniques for the Trace-Driven Simulation of Cache Performance," in *Proc. 1989 Winter Simulation Conference*, Dec. 1989, pp. 1042-1046.

[7] M. Dubois, F. A. Briggs, I. Patil and M. Balakrishnan, "Trace-Driven Simulations of Parallel and Distributed Algorithms in Multiprocessors," in *Proc. 1986 International Conference on Parallel Processing*, Aug. 1986, pp. 909-915.

[8] R. G. Covington, S. Madala, V. Mehta, J. R. Jump and J. B. Sinclair, "The Rice Parallel Processing Testbed," in *Proc. 1988 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Santa Fe, NM, May 1988, pp. 4-11.

[9] R. M. Fujimoto, "The Simon Simulation and Development System," in *Proc. 1985 Summer Computer Simulation Conference*, July 1985, pp. 123-128.

[10] M. J. Flynn, C. L. Mitchell and J. M. Mulder, "And Now a Case for More Complex Instruction Sets," *Computer*, Vol. 20, No. 9, pp. 71-83, Sept. 1987.

[11] E. Williams and F. Bobrowicz, "Speedup Predictions for Large Scientific Parallel Programs on Cray X-MP-Like Architectures," in *Proc. 1985 International Conference on Parallel Processing*, Aug. 1985, pp. 541-543.

[12] R. G. Covington, "Validation of Rice Parallel Processing Testbed Applications," Ph.D. Dissertation, Electrical and Computer Engineering Department, Rice University, Dec. 1988.

[13] R. G. Covington, J. R. Jump and J. B. Sinclair, "Cross-Profiling as an Efficient Technique in Simulating Parallel Computer Systems," in *Proc. IEEE 13th Ann. Int. Computer Software and Applications Conf.*, Orlando, FL, Jan. 1989, pp. 75-80.

[14] R. M. Fujimoto and W. B. Campbell, "Direct Execution Models of Processor Behavior and Performance," in *Proc. 1987 Winter Simulation Conference*, Dec. 1987.

[15] M. Huguet, T. Lang and Y. Tamir, "A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements," in *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, MN, July 1987, pp. 14-25.

[16] B. A. Delagi, N. Saraiya, S. Nishimura and G. Byrd, "An Instrumented Architectural Simulation System," Rept. No. KSL 86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, Jan. 1987

[17] B. A. Delagi, N. Saraiya, S. Nishimura and G. Byrd, "LAMINA: CARE Applications Interface," Knowledge Systems Laboratory, Computer Science Department, Stanford University, Jan. 1987

[18] R. G. Covington and J. R. Jump, "CSIM 2.0 Users Guide," Tech. Rpt. #TR8501, Electrical and Computer Engineering Department, Rice University, Feb. 1986

[19] S. Dwarkadas, J. R. Jump and J. B. Sinclair, "Efficient Simulation of Cache Memories," in *Proceedings of the1989 Winter Simulation Conference*, Washington, D.C., Dec. 1989, pp. 1032-1041.

[20] S. Madala, "Concurrent C Users Manual," Tech. Rept. #8701, Electrical and Computer Engineering Department, Rice University, Oct. 1987

[21] B. N. Bershad, E. D. Lazowska and H. M. Levy, "PRESTO: A System For Object-Oriented Parallel Programming," Tech. Rept. #87-09-01, Computer Science Department, University of Washington, Sept. 1987

[22] P. J. Weinberger, "Cheap Dynamic Instruction Counting," *Bell Systems Technical J.*, Vol. 63, No. 8, pp. 1815-1826, Oct. 1984.

[23] D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, Vol. 1, No. 2, pp. 19-42, Apr. 1984.

[24] S. C. Ingels, "Validation of the Rice Parallel Processing Testbed Using Sorting Algorithms," M. S. Thesis, Electrical and Computer Engineering Department, Rice University, April 1989.

[25] V. Mehta, "Performance Prediction of Fast Fourier Transform Algorithms on Loosely Coupled Multiprocessors," M. S. Thesis, Electrical and Computer Engineering Department, Rice University, May 1988.

[26] G. M. Lauderdale, "Performance Prediction of Packet-Switched Multistage Interconnection Networks in an Execution-Driven Environment," M. S. Thesis, Electrical and Computer Engineering Department, Rice University, Sept. 1989.