Parallel Simulation of Multiprocessor Systems Using Reverse Execution

Kemin Yang and Richard Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{kemin,fujimoto}@cc.gatech.edu

IBM Technical Contacts:
Ram Rajamony and Ahmed Gheith

Abstract

This project is exploring the use of parallel discrete event simulation techniques to perform instruction-level simulations of large-scale multiprocessor systems. The large memory and computational requirements of such simulations and the inherent parallelism of the problem domain suggest the use of parallel simulation techniques. However, a critical challenge lies in ensuring proper synchronization of the parallel computation. Optimistic synchronization techniques such as Jefferson's Time Warp algorithm offer promise to addressing this issue, but have large memory requirements because past state information must be saved to allow one to roll back the computation. We propose the use of reverse execution techniques to greatly reduce the memory required to perform optimistic parallel simulations by re-computing past state information as needed, rather than using traditional state saving mechanisms. This paper describes preliminary work to assess the feasibility of this approach. Specifically, key aspects of the approach are described, as well as initial thoughts on how the generation of reverse execution code can be automated.

1. Introduction

Simulation plays a critical role at all levels in the design and analysis of multiprocessor systems, including assessment of hardware designs, architectures, operating systems and applications. Alternatives to simulation include analytical modeling and hardware prototyping. However, analytical models often are not able to capture important details of the operation of complex multiprocessor systems, and hardware prototypes are expensive to develop and difficult to modify once they have been constructed. Simulation, at least in principle, allows one to circumvent these difficulties, and study the behavior of the system at different levels of detail.

Simulation methodologies used for processor design and evaluation include trace driven simulation, distribution driven simulation, instruction level simulation, and direct execution simulation. Here, the *target processor* refers to the computer one is trying to simulate, and the *host processor* refers to the computer executing the simulation. Trace driven simulation uses trace data generated by executing a benchmark program on some machine, and recording the sequence of machine instructions that were executed. It can only be used to simulate architectures that are similar to the one corresponding to the trace, and is difficult to extend to the simulation of parallel computers [1]. Distribution driven simulation uses statistical approximations of program behavior to drive the simulation. The results can be inaccurate [1]. Instruction level simulation uses a host computer to emulate in detail the effect of executing instructions on a target computer. Detailed information concerning the behavior of the target system can be obtained. However, a number of host instructions must be executed to emulate a single target instruction. Direct execution simulation is a variation on instruction level simulation where one executes most of the machine language instructions of the program on the host directly while simulating only the missing features of the target system on the host. It can produce performance predictions that are nearly as accurate as those obtained with instruction level simulation with much lower overhead [2]. For these reasons, many current simulations of multiprocessor system are based on direct execution.

Direct execution multiprocessor simulations typically run benchmark programs on 'smaller' host machines that have less computation power and/or memory capacity than the target system. Simulators run on a

single processor host machine will not be able to simulate large scale multiprocessor systems due to memory and/or execution time constraints. For this reason, methods of executing instruction level simulations efficiently on parallel host machines are of great interest.

The execution of discrete event simulation programs on parallel machines requires a synchronization mechanism to ensure that the parallel execution yields the same results as a sequential execution. Synchronization mechanisms can be categorized as *conservative* or *optimistic* based on whether transient incorrect computations are allowed to occur during the execution [3]. Conservative simulation does not allow such incorrect computations [4]. Blocking mechanisms are used to ensure transient errors do not occur. The efficiency of conservative parallel simulation is greatly impacted by the amount of blocking that must occur during the execution. This, in turn, is affected by how the simulator identifies what are the "safe" events, and fundamentally, by the amount of *lookahead* in the target system. Lookahead specifies the minimum amount of simulation time in the future that events must be scheduled. It is well known that poor lookahead will result in high synchronization overheads, and serious performance degradations. This can be problematic in tightly coupled multiprocessor systems where interactions between processors may occur with little simulation time delay.

Optimistic synchronization permits transient incorrect computations to occur, but provides a rollback mechanism to erase them [5]. Optimistic execution reduces the reliance on lookahead for achieving efficient execution. Typically, state saving (checkpointing) is used to realize rollback. Using state saving, the original value of a program variable is saved before it is modified by the event computation. Upon rollback, the previously saved state is restored. State saving incurs two principal problems: the amount of memory required to store saved values, and the amount of time required to perform the state saving. Both factors can degrade performance – a large memory footprint reduces the efficiency of the memory hierarchy, and can lead to significant performance degradations. State saving overhead is especially important for programs containing a small amount of computation per event because the state saving operations require a significant amount of time relative to the amount of time spent on simulation computations [3].

One approach to addressing these problems is to use reverse computation rather than state saving [13]. Here, rollback is realized by performing the inverse of the individual operations that are executed in the event computation. Reverse computation attempts to reduce the overhead incurred during the forward execution of the computation, possibly at the cost of increased overhead in the reverse computation path.

Here, we explore the use of optimistic synchronization using reverse execution for multiprocessor simulation. Section 2 describes related work in the parallel simulation of multiprocessor systems. Section 3 describes an approach using optimistic simulation with reverse computation. Section 4 describes future work.

2. Related Work

Many instruction level simulation tools have been developed, with several focusing on execution on parallel host computers. The WWT (Wisconsin Wind Tunnel) is a parallel simulator developed for evaluating cache coherent, shared memory multiprocessors [6]. It directly executes target program instructions and memory references that hit in the target cache, resulting in efficient execution of the simulation of individual processors. WWT uses a conservative time bucket mechanism to synchronize the simulations of the processor nodes. Inter-processor interactions are managed by dividing the program execution into lock step quanta. In the absence of a simulation of the interconnection network, the lookahead becomes the minimum latency in the interconnection network of the target machine. Typically, a fixed network latency is used that determines the time quanta, and thus the parallel simulator's lookahead. The WWT depends on the host machine's hardware to trigger the simulator and synchronize the execution. In the current version, each host processor simulates only one target processor.

Versions of WWT using optimistic synchronization and state saving were also developed, but yielded poorer performance than the version using conservative execution for applications with relatively good lookahead (large time quanta) [7]. Optimistic processing overheads coupled with fast, hardware support for barriers and reductions (which favor conservative synchronization algorithms) were cited as underlying reasons behind these results. The authors speculate that optimistic processing may yield better results when detailed network simulations are required (reducing lookahead) and/or

the host computer has higher message latencies, as is the case in (for example) cluster computing environments.

LAPSE (Large Application Parallel Simulation Environment)'s primary goal is to support scalability and performance analysis of the Intel Paragon machine. It runs on message-passing architectures. Because in a message passing setting only an explicit call to a message passing subroutine can influence network behavior, better lookahead is available and a less rigid synchronization approach is used [8]. The conservative synchronization mechanism used in LAPSE is a variation on the YAWNS protocol using appointments [9].

COMPASS (Component-Based Parallel System Simulator) is an optimistic parallel multiprocessor simulator developed at UCLA using direct execution and parallel discrete event simulation techniques. It provides performance predictions for parallel programs using the MPI message passing library [15]. COMPASS use compile-time code analysis to identify portions of the code that can be removed without invalidating the program, thereby reducing memory requirements. Removed code was pre-executed to determine its execution time so that the simulation time clock could be properly updated when execution reached the deleted code.

MPI-Sim is another parallel simulator developed at UCLA that also focuses on direct execution of MPI-based parallel programs [11]. MPI-Sim uses a conservative execution paradigm, and utilizes compiler analysis of the program to enhance the lookahead of the parallel simulator [18].

SimOS is a system simulator that simulates both application and system level code [19]. SimOS does not use a synchronization protocol, however, when running multiple simulation processes on a parallel host machine, reducing the accuracy of the simulation [20].

A parallel version of the Proteus simulator was developed at MIT for simulating message-based multicomputers [21]. This system uses a conservative synchronization mechanism using optimizations for barrier mechanisms to reduce synchronization overheads.

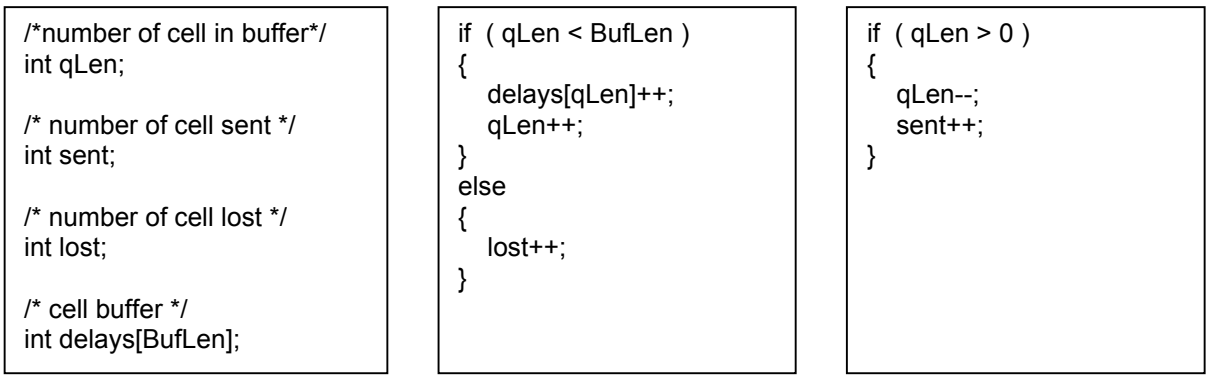
Timepatch is a technique for parallel simulation of cache-coherent multiprocessors. It is based on the fact that the functional correctness of the simulation can be decoupled from timing correctness. Application specific knowledge is used to yield a mechanism that is conservative with respect to functional correctness but optimistic with respect to the timing information. More application parallelism can be exploited without being constrained by conservative synchronization mechanisms since strict timing correctness is not necessarily to be guaranteed. Time inconsistencies are fixed at specific synchronization points [10].

3. Parallel Simulation Approach

As noted earlier, the performance of conservative parallel simulation depends heavily on lookahead, which is determined by the target system and application. Further, synchronization operations can account for a large portion of the execution time. For example, synchronization represents a significant overhead in WWT. Quantum synchronization accounts for as much as 71.3% of the simulation time when simulating the program Ocean on a parallel CM-5 Machine with hardware support for synchronization [11]. This motivates us to examine optimistic solutions to avoid the high cost of synchronization.

The principal challenge in using optimistic execution is to reduce the associated overheads. Two important overheads are those associated with rollback and state saving. Concerning the latter, one study showed that state saving resulted in little or no increase in speedup as the number of processors is increased for applications with a small event granularity on a highly optimized Time Warp system, even when the applications had few rollbacks [12]. Other work by Poplawski and Nicol (1998) also showed that a conservative simulator resulted in significantly better performance than a Time Warp simulator when run on a CC-NUMA architecture for small event granularity applications. As discussed earlier, state saving overhead is problematic for such simulations.

Reverse computation involves computing the previous state of a process rather than checkpointing the state. This involves executing the inverse of individual operations in the event computation to realize rollback, thereby substantially reducing the overheads associated with forward execution. A principal advantage of this approach is that it can substantially reduce the amount of information that must be saved compared to state-saving techniques. The overhead introduced during the forward computation is significantly reduced because only a small portion of code will be added to the application. In [13] the author observed almost six



(a) State (b) Cell arrival (c) Cell transfer

Figure 1: A simple ATM multiplexer model

times faster execution using reverse execution compared to state saving on a commercial CC-NUMA multiprocessor; a key factor in that study was the number of TLB misses were much larger when using state saving because the computation required much more memory.

Our objective is to create a parallel simulation of a PowerPC-based multiprocessor system using optimistic, reverse execution techniques. The parallel simulator consists of a collection of logical processes that interact by exchanging time stamped messages. Here, the simulation of each PowerPC CPU is mapped to a single logical process, and interprocessor communications (e.g., accesses to shared memory) are mapped to messages exchanged among the logical processes. Here, we focus on the simulation of each CPU.

Given a PowerPC assembly language program, two new versions of that program must be generated to implement reverse execution. The first is the original program augmented to create a reversible version of the original code. As will be seen momentarily, some information must be logged during the forward execution to enable the program's execution to later be undone. The second program is the reverse code itself, i.e., the code that is executed to undo the execution of the original code. The forward execution of the optimistic simulator will execute the reversible version of the program. A rollback will trigger the execution of the reverse code.

3.1. An Example

We will now illustrate how reverse computation can be applied to roll back the state of an instruction level simulation of the PowerPC. Rules to automatically generate the assembly language reversible and reverse code are discussed later. We use the example program written in the C programming language described in [2], a simple model for an ATM multiplexer, to illustrate the reverse computation approach. The state of the model is shown in Figure 1(a). Two code fragments, the cell arrival and the cell transfer event handler procedures, are shown in Figures 1(b) and 1(c), respectively [14]. The corresponding PowerPC assembly code is shown in Figure 2. For details about the PowerPC instructions see [16,17]. Our method is not specific to the PowerPC instruction set nor programs originally written in C, and can easily be applied to other machine architectures and languages.

To generate the reversible (forward execution) code we must record the outcome of the branch instructions so the reverse code can determine which instructions must be undone. For this purpose, two additional variables, b1 and b2, are introduced. As shown in Figure 3, we inserted code to set and store the value of b1 and b2 following each branch instruction, and at the branch target to record whether or not the branch was taken. We only need 1 bit to save each variable (b1 or b2). These are the necessary additional operations to make the code reversible.

More generally, we can divide an application's code into functions and each function into basic blocks. Each conditional branch instruction divides the

```
_arrival:
  lwz   r4,  _qLen;   #load qLen to r4
  cmpwi cr0, r4, 255;
  bgt   _greater;     #qLen >= BufLen (=256)?
  lwz   r5,  _delays;
  lwzx  r6,  r5, r4;   #load delays[qLen] to r6
  addi  r6,  r6, 1;    #delays[qLen]++
  stwx  r6,  r5, r4;
  addi  r4,  r4, 1;    #qLen++
  stw   r4,  _qLen;
  b     _ret;
_greater:
  lwz   r7,  _lost;    #load lost to r7
  addi  r7,  r7, 1;    #lost++
  stw   r7,  _lost;
_ret:
```

(a) Cell arrival

```
_transfer:
  lwz   r4,  _qLen;    #load qLen to r4
  cmpwi cr0, r4, 0;
  ble   _ret;          #qLen > 0?
  subfic r4, r4,  1;   #qLen--
  stw   r4, _qLen;
  lwz   r5, _sent;
  addi  r5, r5,  1;    #sent++
  stw   r5, _sent;
_ret:
```

(b) Cell transfer

Figure 2: Machine code of the ATM multiplexer model

```
_arrival:
  lwz   r4,  _qLen;   #load qLen to r4
  cmpwi cr0, r4, 255;
  bgt   _greater;     #qLen >= BufLen (=256)?
  xor   r8,  r8, r8;    #set b1 = 0
  stb   r8,  _b1;       #store b1
  lwz   r5,  _delays;
  lwzx  r6,  r5, r4;   #load delays[qLen] to r6
  addi  r6,  r6, 1;    #delays[qLen]++
  stwx  r6,  r5, r4;
  addi  r4,  r4, 1;    #qLen++
  stw   r4,  _qLen;
  b     _ret
_greater:
  xor   r8,  r8, r8;
  ori   r8,  r8, 1;     #set b1 = 1
  stb   r8,  _b1;       #store b1
  lwz   r7,  _lost;    #load lost to r7
  addi  r7,  r7, 1;    #lost++
  stw   r7,  _lost;
_ret:
```

(a) Cell arrival

```
_transfer:
  lwz   r4,  _qLen;    #load qLen to r4
  cmpwi cr0, r4, 0;
  ble   _ret;          #qLen > 0?
  xor   r8,  r8, r8;     #set b2 = 0
  stb   r8,  _b2;        #store b2
  subfic r4, r4,  1;   #qLen--
  stw   r4, _qLen;
  lwz   r5, _sent;
  addi  r5, r5,  1;    #sent++
  stw   r5, _sent;
_ret:
  xor   r8,  r8, r8;
  ori   r8,  r8, 1;       #set b2 = 1
  stb   r8,  _b2;        #store b2
```

(b) Cell transfer

Figure 3: Reversible code of the ATM multiplexer model

function into 3 basic blocks. Block 1 is the code block before the branch instruction plus the branch instruction itself, block 2 is the code block immediately following the branch instruction, and block 3 is the code starting at the branch target. Code to record the branch outcome is inserted at the start of blocks 2 and 3. More details concerning the automation of this process are discussed in the next section.

Generation of reverse code can be greatly aided by analysis. For example, consider a memory location or register whose value is not used by a block of code before it is modified during the execution of that

```
_r_arrival:
 lbz   r8,  _b1;        #load b1 to r8
 cmpwi cr0, r8, 0;
 bne   _not_equal;  #b1==0?
 lwz   r4, _qLen;
 subfic  r4,  r4, 1;   #qLen--
 lwz   r5,  _delays;
 lwzx  r6,  r5, r4;    #load delays[qLen] to r6
 subfic  r6,  r6, 1;   #delays[qLen]--
 stwx  r6,  r5, r4;
 b     _ret;
_not_equal:
 lwz   r7,  _lost;      #load lost to r7
 subfic  r7,  r7, 1;   #lost--
 stw   r7,  _lost;
_ret:
```

(a) Reverse cell arrival

```
_r_transfer:
 lbz   r8,  _b2;        #load b2 to r8
 cmpwi cr0, r8, 0;
 bne   _ret;            #b2==0?
 lwz   r5, _sent;
 subfic  r5, r5,  1;    #sent--
 stw   r5, _sent;
 lwz  r4, _qLen;
 addi r4, r4,  1;       #qLen++
 stw   r4, _qLen;
_ret:
```

(b) Reverse cell transfer

Figure 4: Reverse code of the ATM multiplexer model

block of code. There is no reason for the reverse computation code to restore the values of such variables. Storage used for temporary values (e.g., registers) fall into this category.

Reverse code for the two procedures are shown in Figure 4. Here, it is assumed registers are used as temporary storage, so their state need not be restored by the reverse execution code. Because the branch information was recorded during the forward computation path, the reverse code need only examine the saved branch information to determine which blocks of code must be reverse executed. The reverse code is determined by identifying the state information that must be restored, analyzing the instructions that modified that state, and executing in reverse order the inverse operations for those instructions. An instruction's inverse is the instruction(s) needed to restore the program state to that which existed prior to the instruction's execution. In the above example, an addition instruction's inverse instruction is a subtract instruction. Some instructions may have no inverse instruction, for example, the comparison instructions, while other instructions may need multiple instructions to roll back their effects, for example, the logical instruction AND, OR etc.

Compared to state saving, it is clear that the reverse computation approach requires much less memory. Only 2 bits are required in the above example while the simple copy state saving technique will need BufLen+3 words. Even if incremental state-saving techniques are applied to this model, several words are still needed to save the changed data values. At the same time, the overhead in the forward computation path is reduced because only 2 bits are saved.

3.2. *Automation*

We now consider the problem of automatically generating reversible and reverse code. The general process for reverse code generation includes the following steps: (1) identify functions and code blocks, (2) construct a code block calling graph, (3) record forward execution path and (4) generate reverse code.

Identify Functions and Code Blocks

Functions are the basic building units of a program. A function can only execute code of another function by calling it. At the assembly language level, because every processor architecture has a 'function pattern' based on the certain calling conventions, it is straightforward to identify each function.

A code block (basic block) is a set of instructions that are always executed in sequence, and execution may only enter the block at the first instruction and leave

```
----------------------code block 1--------------------
_arrival:
  lwz   r4,  _qLen;   #load qLen to r4
  cmpwi cr0, r4, 255;
  bgt   _greater;     #qLen >= BufLen (=256)?
----------------------code block 2--------------------
  lwz   r5,  _delays;
  lwzx  r6,  r5, r4;    #load delays[qLen] to r6
  addi  r6,  r6, 1;     #delays[qLen]++
  stwx  r6,  r5, r4;
  addi  r4,  r4, 1;     #qLen++
  stw   r4,  _qLen;
  b     _ret;
----------------------code block 3--------------------
_greater:
  lwz   r7,  _lost;     #load lost to r7
  addi  r7,  r7, 1;     #lost++
  stw   r7,  _lost;
----------------------code block 4--------------------
_ret:
```

Figure 5: code blocks of the cell arrival function

at the last instruction of the block. The reason to divide a function code into smaller code blocks is to record the forward execution path. Code blocks are divided by branch and jump instructions. Each branch instruction (except the function return instruction) adds two separate lines to a function code, the first line immediately follows the branch instruction, and the second precedes the branch target. A code block is the set of instructions between two adjacent separate lines. For example, the machine language code of the arrival event handler has 4 code blocks, as shown in the figure 5.

Code blocks in the same function are assigned unique IDs after they have been identified. As a convention, the first code block will be assigned ID 1, the second code block will be assigned ID 2, etc.

Construct Code Blocks Calling Graph

Once code blocks are identified, we need to know the order of execution among them. The execution order of code blocks can be represented by a directed graph, the calling graph. Each node of the graph represents a code block and keeps the code block's information: the code block ID, address of the last instruction, and the number of instructions in the code block which will be used to locate the code block. If a code block B can be executed immediately after another code block A, an arc is created from A to B. The graph can be generated statically by parsing the assembly code.

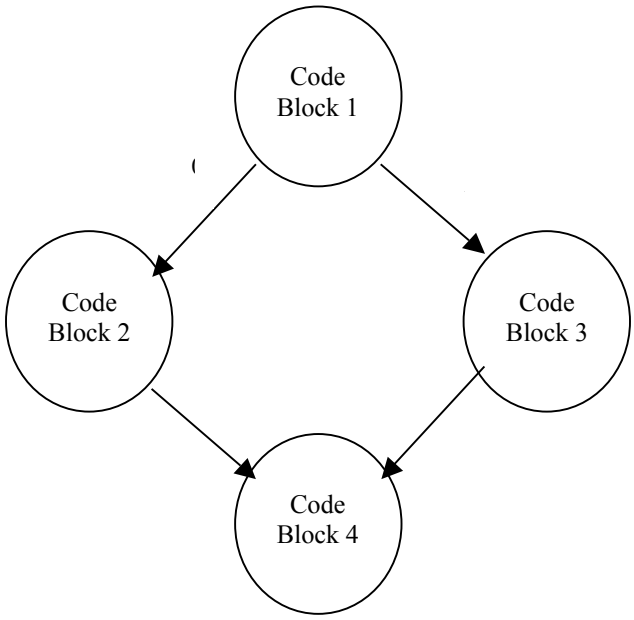


Figure 6: Calling graph, cell arrival function

Record Forward Execution Path

In the calling graph for the assembly code, a node can have at most two outgoing arcs. The arc to the neighbor with the smaller code block ID is labeled 0, and the other arc is labeled 1. The calling graph of the cell arrival event handler is shown in Figure 6.

The forward execution path must be recorded for reverse execution. This can be accomplished by recording the sequence of executed code blocks in the calling graph. We use a stack to record the execution path. Each time a function finishes execution of a code block, it will take an arc in the calling graph to execute the next code block. The label of the corresponding arc is pushed on the stack.

Generate Reverse Code

Reverse execution requires executing the inverse instructions of code blocks in the reverse order from the forward execution path. The calling graph and the forward execution record are used to determine which code block should be executed next in the reverse execution. When a reverse function finishes reverse executing a code block, it checks if there are two incoming arcs go to the block. If there are, a bit is popped from the forward execution path record stack to decide which reverse path should be taken.

The goal of the inverse instructions is to restore the program state to that which existed before the execution of the code block. Here program state refers to the memory state. To accomplish this, we need to identify whether and how an instruction potentially affects the memory state.

Many instructions have simple inverse instructions. For example, in the PowerPC instruction set, the inverse of an addition instruction where one of the source registers is also a destination register is simply a subtraction instruction. We call this type of instruction 'constructive instructions'. Some instructions have no simple inverse instruction, for example, most of the logical instructions such as AND, OR etc. We called this type of instruction 'destructive instructions'. Other instructions which don't affect the state of the general purpose registers, such as the comparison instructions, branch instructions etc., can be simply ignored.

To generate reverse code for a function, we first determine all the instructions that can directly change the memory state. For the PowerPC instruction set, only store instructions change the memory state directly.

A store instruction overwrites a memory location with a new value. To rollback to the previous memory state, we need to determine the old value in the memory location and restore this value. In many cases, the old value can be found in, or derived from, the instructions in the single function code body by tracing backward along the forward execution path. For the cell arrival function in the above example, the old value in register *r6* in the store instruction *'stwx r6, r5, r4'* can be derived from the instructions *'addi r6, r6, 1'* and *'lwzx r6, r5, r4'*.

The following steps are used to determine the old value: First, the register *rs* holding the new value in the store instruction is identified. Then we trace backward along the forward computation path to find all instructions that may change the value of *rs* until a load instruction is found. It is possible there are multiple load instructions along the backward path, however, the above process only stops at the first one that might change the value of *rs*. The other load instructions can be ignored. After that we compare the memory address operand(s) of the load and the store instruction. If the memory operands hold the same values, the register operand of the load instruction holds the old value.

If the instructions along the backward path from the store instruction to the load instruction which may change the value of *rs* are constructive instructions, then we can generate the reverse code by replacing each instruction with its inverse instruction. The store instruction will be replaced by a load instruction which loads the new value into *rs* and the load instruction will be replaced by a store instruction which stores the old value into the memory address.

When no corresponding load instruction can be found for a store instruction or there are destructive instructions along the backward path, the old value can't be recovered. In these cases, we can use incremental state saving to recover the old value.

4. Future Work

There are many open issues remaining to be resolved. An assessment of the performance that can be obtained using this approach is needed, and its merits relative to conservative synchronization methods and optimistic execution using state saving mechanisms is required. For this purpose, we plan to develop a parallel simulation framework that is able to support multiple synchronization mechanisms, including optimistic simulation using reverse execution. Initially, reverse execution code will be generated manually. This implementation will be based on the PowerPC instruction set. A subset of the Mambo simulator will be utilized as the starting point for this effort.

A compiler to automatically generate more efficient reverse execution code is another open area of research. Techniques to optimize the performance must be developed. For example, the size of the code block graph can be reduced by defining methods to combine multiple tightly coupled code blocks into a single block. One does not necessarily have to resort to state-saving to inversely execute each destructive instruction. Automating the generation of more efficient reverse execution code is clearly required to make theis technique practical.

5. References

[1] R.G. Covington, et al. *"The Efficient Simulation of Parallel Computer System",* International Journal in Computer Simulation, Vol.1 1991.

[2] R. G. Covington, S. Malada, V. Mehta, J. R. Jump and J. B. Sinclair, *"The Rice Parallel Processing Testbed",* in Proc. 1998 ACM SIGMETRICS Conf.

on Measurement and Modeling of Computer Systems, 1988.

[3] Richard M. Fujimoto, *"Parallel and Distributed Simulation Systems",* Wiley & Sons, 2000.

[4] K. M. Chandy and J. Misra. *"Asynchronous distributed simulation via a sequence of parallel computations",* Comm. Of the ACM, Vol. 24, No. 11, Nov. 1981.

[5] D. A. Jefferson. *"Virtual time",* ACM TOPLAS, Vol. 7, No. 3, Jul. 1985.

[6] Steven K. Reinhardt et al., *"The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers",* ACM SIGMETRICS, 1993.

[7] Sashikanth Chandrasekaran and Mark. D. Hill, *"Optimistic Simulation of Parallel Architectures Using Program Executables",* in 10th Workshop on Parallel and Distributed Simulation (PADS96), 1996.

[8] Philip M. Dickens etc, *"Parallelized Direct Execution Simulation of Message Passing Parallel Programs",* Technical Report 94-50, ICASE, July 1994.

[9] D. M. Nicol. *"Parallel Discrete-event Simulation of FCFS Stochastic Queuing Networks",* In Proceedings ACM/SIGPLAN PPEALS 1988.

[10] Umakishore Ramachandran, Gautam Shah, Ivan Yanasak and Richard Fujimoto. *"Timepatch: A Novel Technique for the Parallel Simulation of Multiprocessor Caches",* Technical Report GIT-CC-96-24, College of Computing, Georgia Institute of Technology

[11] S. Prakash, E. Deelman, and R. Bagrodia, "*Asynchronous Parallel Simulation of Parallel Programs*," IEEE Transactions on Software Engineering, Vol. 26, 2000.

[12] C. Carothers, K. S. Perumalla and R. M. Fujimoto. *"The Effect of State-saving in Optimistic Simulation on a CACHE-coherent Non-uniform Memory Access Architecture",* 1999 Winter Simulation Conference Proceedings, 1999.

[13] C. Carothers, K. S. Perumalla and R. M. Fujimoto. *"Efficient Optimistic Parallel Simulations using Reverse Computation",* In Proceedings of 13th Workshop on Parallel and Distributed Simulation, May 1999.

[14] K.S. Perumalla and R. M. Fujimoto. "*Source code transformations for efficient reversibility*." *Technical Report, GIT-CC-99-21*, College of Computing, Georgia Institute of Technology.

[15] Thomas Phan and Rajive Bagrodia, *"Optimistic Simulation of Parallel Message-Passing Applications",* Workshop on Parallel and Distributed Simulation, 2001.

[16] A. Marsala, *"The PowerPC Architecture: A Programmer's View",* IBM, 2001.

[17] Motorola Inc. *"PowerPC Microprocessor Family: The Programming Envrionments for 32 Bit Microprocessors",* 1997.

[18] Ewa Deelman, et al., "*Improving Lookahead ion Parallel Discrete Event Simulations of Large-Scale Applications Using Compiler Analysis*," Workshop on Parallel and Distributed Simulation, 2001.

[19] M. Rosenblum et al., "*Complete Computer System Simulation: The SimOS Approach*," *IEEE Parallel and Distributed Technology*, vol. 3, 1995.

[20] M. Rosenblum, et al., "*Using the SimOS Machine Simulator to Study Complex Computer Systems*," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, 1997.

[21] U. Legedza and W. E. Weihl, "*Reducing Synchronization Overhead in Parallel Simulation*," Workshop on Parallel and Distributed Simulation, 1996.