Shared Memory as a Basis for Conservative Distributed Architectural Simulation [1]

Mark R. Swanson
Leigh B. Stoller
E-mail: {swanson,stoller}@cs.utah.edu

UUCS-97-005

Department of Computer Science
University of Utah

Abstract

This paper describes experience in parallelizing an execution-driven architectural simulation system used in the development and evaluation of the Avalanche distributed architecture. It reports on a specific application of conservative distributed simulation on a shared memory platform. Various communication-intensive synchronization algorithms are described and evaluated. Performance results on a bus-based shared memory platform are reported, and extension and scalability of the implementation to larger distributed shared memory configurations are discussed. Also addressed are specific characteristics of architectural simulations that contribute to decisions relating to the conservatism of the approach and to the achievable performance.

1 Introduction

Architectural simulation is a valuable tool in the processes of exploring, designing, enhancing and implementing computer systems. *Execution-driven* simulation is especially valuable in gaining an understanding of an architecture in a dynamic sense, in exploring the interaction of its parts, and in evaluating macro and micro performance characteristics. In addition, it enables the implementation of working software based on that architecture, allowing software costs, including programming, to be evaluated concurrent with architectural design.

Accurate architectural simulation is costly, with slowdown factors of 10's to 1000's being reported for simulations of uniprocessors. When simulating multiprocessors, as the Avalanche group is doing, the simulation process is generally slowed down by a factor proportional to the number of processors simulated, by the additional simulation of the interconnect, and by context switching within the simulator between the simulated processors.

[1]This work was supported by a grant from Hewlett-Packard, and by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

For research groups developing parallel architectures, exploiting parallelism to speed up the required architectural simulations is an obvious approach. This paper reports on such a parallelization effort and its unusual approach of performing distributed simulation within a shared memory model.

1.1 The Avalanche Architecture

The Avalanche distributed system will be a cluster or network of 32 to 64 workstations[7] interconnected with a Myrinet network. Its unique aspects lie in providing a communications interface supporting extremely efficient message passing and distributed shared memory (DSM) and designing that interface to plug in to commodity workstations. All interactions between processors occur as communications over the Myrinet via this interface. The communications are always initiated by processor memory references: stores to interface registers in the case of message passing and cache misses in the case of DSM. The design of the Avalanche interface is intended to minimize all aspects of interprocessor communication overhead, with special emphasis on memory hierarchy effects. To expose these effects, a detailed and quite fine grained simulation is required.

2 The Base Simulation Environment

The base uniprocessor simulation environment developed by the Avalanche project is comprised of a simulator for the HP PA-RISC architecture[5], including an instruction set interpreter, and detailed simulation modules for the first level cache, the system bus, the memory controller, the network interconnect, and the communications device which is the focus of the Avalanche project's research. This environment is called PAint (PA-interpreter)[6] and is derived from the Mint simulator[9]. The simulator is designed to model multiple nodes, consisting of the modules listed above, and the interactions between nodes, with emphasis on the effects of communication on the memory hierarchies.

PAint schedules *tasks* to perform simulation events at specified times in the future–with the present time being an acceptable degenerate case. Consider modeling a first level cache access. When an instruction performs a memory reference, a task is scheduled to model the resulting cache activity. This task would first need to look up the access in the tag rams. To model the time cost of this, it schedules itself to resume after an appropriate delay. When the task is again executed, it performs the tag lookup, and assuming it succeeds, once again schedules itself in the future to model the delay involved in accessing the data rams and returns to the scheduling loop. When contention arises for a resource, such as the tag rams, the task requiring access to that resource will enqueue itself on the resource, and the holder of the resource will schedule that task to execute when the holder finishes with the resource. It is thus possible that several tasks may be scheduled concurrently for a particular node, executing both overlapped and interleaved in time, depending on the interactions of the modeled entities. The computational granularity of these tasks varies widely, from one or two microseconds to 30 or more microseconds (on the platform discussed in this paper).

Current simulation time in PAint is the minimum time of the tasks in the task list[2]. Scheduling a task is accomplished by *inserting* it in a list ordered by desired execution time; executing a task involves *extracting* the task with the lowest execution time from the list and performing the action described by the task. In sequential PAint, there is but a single task list for all simulated nodes, so

[2]The data structure used is actually an array of lists. It's implementation is not crucial to the results described.

simulation of nodes is interleaved. A representative time to insert a task on the platform described here is 775 nanoseconds; representative extraction times range from 700 to 1200 nanoseconds.

The driving task in all of this is the instruction execution task, modeled by the `next_event` function. It interprets instructions which ultimately initiate actions in all of the other modules. PAint is an example of an execution-driven simulator. It interprets *icodes*, a pre-decoded form of instructions. The decoding is performed at program load time. The `next_event` task interprets icodes until it reaches one that potentially results in actions in other modules within its processor and possibly in other processors. Such an icode is preceded by a special *event* icode inserted by the loader. This event breaks the task out of the interpretation loop, suspends the `next_event` task, and causes the scheduling of some other task, such as the cache task described earlier. In PAint, the most common events of interest are memory references.

Since the `next_event` task may iterate through several instructions each time it is scheduled, the advance of its time is "chunky." That is, its time will have advanced several machine cycles as a result of being executed once, and it may be "ahead" of other tasks. This does not result in causality errors, since it always reschedules at points where external entities, such as other processors, might influence it.

3 Parallelizing the Simulation

The motivation for developing PPAint (Parallel PAint) was an immediate pragmatic one of needing to run larger simulations and run them more quickly. The simulator is a complex program, tuned for uniprocessor performance. It was not desirable to either increase its complexity significantly, nor to decrease its uniprocessor performance in order to parallelize it, nor was a long term parallelization effort possible.

A shared memory approach was chosen for three reasons:

1. **synchronization costs**: *at present*, basic communication costs between processors in message passing systems are between one and two orders of magnitude greater than those in shared memory systems. It appeared likely that these costs on a message passing platform would dwarf the actual computation, resulting in little or no speedups.

2. **complexity**: The software cost of building efficient, message based synchronization is also higher than the methods used in the work reported here, which are based on shared variables.

3. **availability**: an SGI Power Challenge system was available and acquisition of Hewlett-Packard shared memory multi-processor workstations is planned. In the future, the Avalanche machine itself, with its distributed shared memory capability, will provide an additional platform.

In spite of targeting shared memory platforms, PPAint resembles a conservatively-synchronized distributed simulation with the simulated nodes acting as the *logical processes* (LPs). The use of shared memory is limited to (1) maintaining global simulation time, (2) inter-processor task scheduling, and (3) providing the substrate for the architectural message passing being modeled.

Particular characteristics of multi-processor simulation motivate this use of a distributed logical process approach and the choice of the node as the LP:

1. The simulation of individual nodes is naturally independent except for well-defined synchronizing events.

2. For many of the applications simulated, the work load per simulated node is very similar, making the node processor an acceptable "unit" for load balancing purposes.

3. Most simulator data structures are naturally created on a per-simulated-node basis and some are shared by components within the node. LP state is most naturally bounded at the node boundary.

4. Within a node, modules schedule tasks in other modules frequently and often do so with zero delay; non-instantaneous interaction is a crucial assumption in many distributed simulation scheduling algorithms. In making the node the LP, these interactions are encapsulated within the LP and are handled by normal sequential scheduling algorithms.

5. Tasks are scheduled between nodes relatively infrequently, and always do so with non-zero delay, enabling use of standard distributed synchronization algorithms.

As a result, PPAint is structured as a collection of slightly modified uniprocessor PAint's, where each PAint models one or more simulated nodes. The modifications are largely isolated to initialization and synchronization code; the component-modeling modules, with only two minor exceptions, remain unchanged.

As expected, this approach proved to be economical in terms of implementation time. From initiation of the project to working parallel simulator was a matter of only a few weeks effort by a single programmer. The subsequent performance improvements and experiments reported here required only a few more weeks.

3.1 Maintaining Causality and Time between Simulated Nodes

Simulated nodes in the architectural model can only affect one another by sending messages, either explicitly in message passing applications or implicitly in the use of DSM. The inherent latency of this message passing, in terms of simulation cycles, is the basic *lookahead* factor used to reduce synchronization between simulation processors. This latency and the resulting lookahead are determined by the nature of the interconnect modeled and the level of accuracy desired from that model. The interconnect modeled here has a 125 nanosecond delay, on-the-wire, for a single-switch fabric. That translates into 15 cycles for simulated processing nodes with a 120 MHz clock. The lookahead for a simulation that ignores contention in the switch is thus 15 cycles. Many interconnects, including the one modeled here, would require flit-by-flit modeling to capture such contention effects. Such fine grained simulation is prohibitively expensive for uniprocessor simulators, but that fine grain also makes attaining speedup from parallel simulation difficult. Such accuracy is sometimes required, of course; making it practical is one of the goals of PPAint development.

At a low level, simulated nodes in PAint actually affect each other through the scheduling of tasks, specifically tasks executing within the network interface model which, in turn, schedules tasks in the memory system model. The network interface can also effect the simulated processor directly by interrupting it. Thus explicit communication between simulation processes is isolated to just a single task scheduling location in the network interface module. The remainder of the simulator, except for initialization code and the task extraction routine, is essentially oblivious to whether it is running as a parallel program or not.

3.2 Time Management

As a distributed simulation, PPAint does not maintain a centralized global clock. Each simulation process maintains its own time in a clock variable globally visible to all simulation processes. Each simulation process also maintains a time value at which it needs to synchronize with the rest of the simulation. The task scheduling loop extracts tasks from the local task list, advancing its clock variable as the schedule time of those tasks moves forward. When it reaches a task that would advance its clock beyond the synchronization time, it invokes a synchronization algorithm which does two things: (1) it computes and returns the time for the next synchronization and (2) dequeues any inter-processor tasks and inserts them into the local task list.

3.3 Inter-Process Task Ordering and Causality

The network module discriminates between tasks it schedules for nodes simulated by its own processor and those it schedules for nodes simulated on other processors. Tasks bound for other processors are placed on a queue owned by that processor; access to that queue is protected by a lock. Correct ordering of task execution is ensured as follows:

- Inter-processor network tasks are constrained to be scheduled with a delay into the future that is greater than the lookahead value.
- A task adding a task to another processor's queue will not advance its own processor's time until the queue operation is visible at the other processor.
- After computing a new synchronization time, the synchronization code always moves all inter-processor tasks from the queue into its local task list before allowing the local clock to advance beyond the old synchronization time.

Maintaining these ordering constraints is trivial in a shared memory system that provides sequential consistency. A store into a global variable is known to be logically visible to the other processors when the store instruction completes. In a system with weaker consistency, it may be necessary to perform a write barrier between an inter-processor queue operation and the subsequent advance of the queuing processor's clock variable.

4 Performance Results with Parallel PAint

It is widely recognized that synchronization and waiting time are the major overheads in conservative parallel simulations. Not surprisingly, the performance of PPAint, over the range of processor counts studied, is significantly impacted by these factors. Several progressively more sophisticated, but still conservative, synchronization algorithms were implemented in an attempt to tolerate these load imbalances.

Another common difficulty for multiprocessor simulations is the frequent synchronization required for accurate simulation of the interconnect. Variable-lookahead variants of the synchronization algorithms were implemented to address this problem.

4.1 Experimental Testbed

All tests were conducted on an SGI Power Challenge with 14 90 MHz R8000 processors and a common memory of 2 gigabytes. Each processor has 16 kilobyte on-chip data and instruction caches and a 4 megabyte unified second level cache. The penalty for a miss to main memory is 53 cycles, while a miss to another cache takes 80 cycles. The simulator was compiled with the MIPS C compiler in 32 bit mode. Micro measurements were obtained using a 21 nanosecond resolution interval timer; an average overhead of reading the timer has been factored out of reported times.

Macro (whole program) times reported were wall clock time as reported by the *getrusage()* system call. The tests were not run on a dedicated machine but simulated processes did not share processors with other user processes.

4.2 Synchronization Algorithms

Figure 1: The BARRIER Time Algorithm

```
mintime = clock[ThisProcessor];
for (i = 0; i < Number_of_rprocs; i++)
    if (clock[i] < mintime)
        return mintime;
return mintime + lookahead;
```

Several synchronization algorithms were evaluated; four are reported on here. The procedures are called from a loop which checks to ensure that the returned time is in its future; if not, it calls the procedure again. The first is algorithm is BARRIER (see Figure 1), in which all processors join a barrier every `lookahead` cycles. This algorithm is simple to implement, using one `clock` variable per processor to hold its simulation time. The simulation time need only be updated once per synchronization epoch, as the processor enters the barrier, minimizing communication time. As other studies have found, this algorithm is prone to high waiting times.

Figure 2: The SIMPLEMIN Time Algorithm

```
mintime = clock[0];
for (i = 1; i < Number_of_rprocs; i++)
        mintime = min(clock[i], mintime);
return mintime + lookahead;
```

SIMPLEMIN (see Figure 2) is a modified version of BARRIER that trades extra communication and more frequent synchronization for waiting time. In SIMPLEMIN, each processor determines the minimum simulation time across all processors; the lookahead value is added to this minimum and, if it is greater than the processor's current time, it continues simulating up to that time. The extra communication cost arises because the processors' global clock variables must be updated on each cycle for this algorithm to be effective. The extra synchronization cost arises because the window between synchronizations is, on average, significantly less than the lookahead value. Two conditions are necessary for SIMPLEMIN to outperform BARRIER: (1) the time to perform a synchronization must be significantly less than the average time spent in simulation between synchronizations, and (2) the cost of communicating the clock values must be low.

TWOWINDOW (see Figure 3) is an adaptive algorithm. Since interaction between processors is confined to network communication, it is possible to identify impending communications some number of cycles before they actually occur. TWOWINDOW computes a minimum *horizon* value over all processors. It is the sum of each processor's time and its current lookahead value, based on whether or not it is, or soon will be, communicating with another processor. Three factors increase synchronization time over the previous algorithms: (1) the added complexity in computing the

Figure 3: The TWOWINDOW Time Algorithm

```
horizon = clock[0] + lookahead[0];
for (i = 1; i < Number_of_rprocs; i++)
        horizon = min(horizon,  clock[i] + lookahead[i]);
return horizon;
```

horizon, (2) the communication of two values: the processor's clock and the current lookahead value, and (3) the dynamic maintenance of the lookahead values. TWOWINDOW can, of course, be generalized to an arbitrary number of lookahead values, limited by the nature of the simulated system and the acceptable complexity in determining and dynamically maintaining those values. Like SIMPLEMIN, it depends on each processor updating its global clock on each cycle.

Figure 4: The CLSTR2WIN Time Algorithm

```
horizon = clock[ThisClusterBase] + lookahead[ThisClusterBase];
for (i = ThisClusterBase + 1; i < ThisClusterMax; i++)
        horizon = min(horizon, clock[i] + lookahead[i]);

Cluster_Horizon[ThisCluster] = horizon;
for (i = 0; i < Number_of_clusters; i++)
        horizon = min( horizon, Cluster_Horizon[i]);
return horizon;
```

CLSTR2WIN (see Figure 4) is a *clustered* minimum calculation. Processors are grouped into clusters. Processors within a cluster use one of the algorithms described above such as SIMPLEMIN or TWOWINDOW across the processors within the cluster. A synchronizing processor then posts its cluster minimum in a per-cluster clock variable. Next it computes a minimum over the cluster clocks. For a system of N processors and a cluster size of M, this algorithm decreases synchronization time and communication from $O(kN)$ to $O(k(M + N/M))$. For the small system sizes reported here, the effects are small, but for larger systems CLSTR2WIN should extend the scalable range of the base algorithm it is applied to. It also can form the basis for clustered clock management in hierarchical, NUMA systems, where both locality and the number of sharers can effect the communication cost of a given shared variable.

Figure 5 shows the time for these basic synchronization operations, for four and eight processor runs. All times were produced with a load of two simulated nodes per processor, each running an SOR calculation. The times reported were gathered for the initial iteration of the algorithm at each synchronization event. This iteration is likely to be the most costly, since it incurs the most communications cost, in the form of cache misses. It is also representative of the minimum cost that a synchronizing processor must pay at each such event. Also reported is an **effective lookahead** metric. This is the average number of cycles a processor is allowed to advance between synchronization events. For the BARRIER case, it is always the lookahead factor, in this case 15.

In evaluating these synchronization times, it is useful to consider the average work performed by one processor in a simulated cycle. In the simulations just reported for 8 processors, the average task time was 10.7 microseconds and on average .45 task was executed every cycle. Consider

Figure 5: Synchronization Operation Timings

Processors	Sync Algorithm	Microseconds per sync	Effective Lookahead
4	BARRIER	5.63	15
4	SIMPLEMIN	5.75	7
4	TWOWINDOW	7.41	13 to 15
4	CLSTR2WIN	9.95	13 to 15
8	BARRIER	9.51	15
8	SIMPLEMIN	10.37	7 to 9
8	TWOWINDOW	13.31	8 to 10
8	CLSTR2WIN	10.35	7 to 9

Figure 6: Synchronization Algorithm Comparison

Sync Algorithm	Runtime Min:secs	Effective Lookahead	Speedup
BARRIER	5:42	15	3.57
SIMPLEMIN	4:06	3	4.96
TWOWINDOW	3:48	4 to 5	5.35
CLSTR2WIN	3:48	4 to 5	5.35

the most expensive synchronization algorithm, TWOWINDOW. With its effective lookahead of approximately 9, the average computation per synchronization would be 43.3 microseconds or about 3.25 times the average synchronization cost.

4.3 Comparative Effectiveness of the Algorithms

Each of the algorithms described above makes a different tradeoff of communication and computation complexity and synchronization frequency. Figure 6 shows the results of tests using 8 real processors, simulating 32 nodes running an SOR program. The basic lookahead time is 15 cycles, while the enhanced lookahead used by the TWOWINDOW and CLSTR2WIN algorithm is 35 cycles. Speedups are calculated based on a uniprocessor version of PAint that ran in 20 minutes 20 seconds.

From these results, the TWOWINDOW-type algorithm (of which CLSTR2WIN is a variant) is a clear winner. This happens despite the fact that the effective window size declines to only 4 to 5 cycles between synchronizations. Even the very simple SIMPLEMIN algorithm performs much better than BARRIER, again in spite of significantly increased synchronization activity.

The real benefit of TWOWINDOW-type algorithms, however, lies in their ability to tolerate the kind of lookaheads required for more accurate network simulations. Figure 7 shows performance as the basic lookahead window is decreased, keeping the enhanced lookahead window constant at 35. The speedup degrades only modestly even with a basic lookahead of 1, which should allow accurate, flit-by-flit modeling of the interconnect. A program that communicates more frequently will see a greater degradation, of course.

Figure 7: Lookahead Tolerance of TWOWINDOW Algorithm

Lookahead	Runtime Min:Secs	Effective lookahead	Speedup
15	3:48	4 to 5	5.35
5	3:56	4 to 5	5.17
2	3:52	4 to 5	5.26
1	3:57	4 to 5	5.15

4.4 Overall Performance

Figures 8 and 9 report speedups obtained for simulations of 3 programs: two successive-over-relaxation programs (SOR-sync and SOR-async) and a gaussian elimination (GAUSS). These are modified versions of programs used by [2]. The modifications consisted of replacing the message passing libraries with ones based on Direct Deposit[8], a protocol suite developed for the Avalanche system. SOR-sync performs two basic kinds of communication at each time step: a global reduction and accumulation and broadcast of a solution vector. The reduction uses a tree rooted at simulated node 0, while the solution vector operation involves all-to-all communication. SOR-async implements the propagation of the solution vector by having each node broadcast values as they are computed; the particular implementation results in a large increase in messages sent and bytes communicated but results in a faster convergence. It still performs the global reduction at each time step. GAUSS performs a global reduction at each step to determine the owner of the pivot row, followed by broadcast of this row from the owner to all the other simulated nodes.

Figure 8: Speedups Using TWOWIN

Simulated Program	Real Processors	Simulated Processors	Runtime	Speedup
SOR-sync	1	32	20:20	
SOR-sync	2	32	11:24	1.78
SOR-sync	4	32	6:19	3.22
SOR-sync	8	32	3:38	5.35
SOR-sync	1	64	53:20	
SOR-sync	2	64	28:51	1.85
SOR-sync	4	64	15:34	3.43
SOR-sync	8	64	9:10	5.82
SOR-async	1	16	19:11	
SOR-async	2	16	10:59	1.75
SOR-async	4	16	6:12	3.09
SOR-async	8	16	3:49	5.02
GAUSS	1	32	40:31	
GAUSS	2	32	25:42	1.58
GAUSS	4	32	15:05	2.69
GAUSS	8	32	9:53	4.1

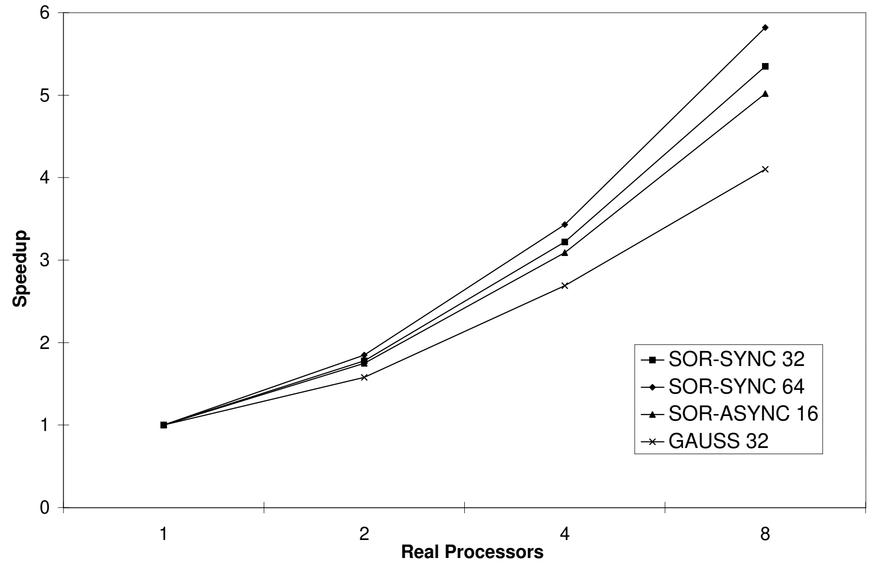

Figure 9: Speedup Curves using TWOWINDOW

The simulator was configured to use the TWOWINDOW synchronization algorithm, using lookahead values of 35 and 15 cycles. The results are much as one would expect. An increase in the number of nodes simulated per processor results in greater speedup. An increase in the amount of communication results in a decrease in speedup; compare SOR-sync against SOR-async and GAUSS, which have greater amounts of communication.

Figure 10 gives results using an enhanced version of TWOWINDOW, which tracks processors that are targets of communication, rather than those that are sources. This results in a more precise application of the appropriate lookahead value. This becomes important with increasing number of simulated nodes and with applications with relatively high communication to computation ratios.

Figure 10: Speedups Using Enhanced TWOWIN

Simulated Program	Real Processors	Simulated Processors	Runtime	Speedup
SOR-sync	8	64	8:50	6.04
GAUSS	4	32	13:45	2.95
GAUSS	8	32	9:00	4.5

5 Scalability

Shared memory systems are frequently criticized as lacking scalability. Emerging DSM systems promise to extend shared memory from the current 4 to sixteen processors on high-performance bus-based systems to hundreds of processors. The high cross sectional bandwidth of switch-based fabrics makes possible this order-of-magnitude increase in processor count. Latency of cache misses due to fabric delays and protocol processing are higher, of course, than in bus-based systems such as the Power Challenge.

The scalability of shared memory distributed parallel simulation such as that reported here remains to be seen. The challenges will be in the areas they have always been in distributed simulation: synchronization and waiting. Synchronization effects should fare no worse for DSM than for message passing systems, since the underlying communications fabrics and transport mechanisms will likely be identical. Arguably, DSM will do better as a synchronization substrate than message passing. The frequent updating of each processor's clock variable is analogous to null messages in a message based system. The variable serves to coalesce sequences of updates so that another processor always get the single latest clock value when it performs a synchronization. This coalescing also serves to reduce bandwidth consumption in the interconnect. For example, the average time to extract a task from the task list when using 8 processors simulating 64 nodes is 750 nanoseconds, which is comparable to the time in a similar uniprocessor run. This indicates that many updates to the clock variables, which occur within the task extraction procedure, are coalesced, costing only as much as a local cache access.

A thornier problem is waiting for slow processors. Efficient implementations of the shared memory model offer the opportunity to trade frequent communication and synchronization for decreased waiting. As long as the fundamental synchronization model remains conservative, of course, temporary local load imbalances will lead to waiting. Short of adopting an optimistic strategy, decreasing the computational weight of individual tasks, to allow finer-grained scheduling, offers the best hope of decreasing waiting time. Little effort was made in this direction in the work reported here; it will be undertaken when availability of a larger platform makes it practical.

6 Related Work

Numerous groups have developed simulators for multiprocessor architectures. A few of them are surveyed here.

The Wisconsin Wind Tunnel[3] uses direct execution of instrumented programs on a CM-5, resulting in very fast and accurate simulation. Interactions between nodes occur only when programs access shared memory locations, which are translated by the underlying simulation system into message passing events between CM-5 nodes. Processor execution proceeds in lock step using a conservative window approach, with control returning to the simulator at the end of the window, or when a non-local memory operation is performed that misses in the cache. The CM-5's fast reduction operators are used to ensure that all processors have reached the end of the current window before proceeding. The main disadvantages of the WWT is the dependence of the simulation environment on the CM-5 hardware and the lack of flexibility in modifying many aspects of the architecture due to its direct execution nature.

Parallel Proteus[1] performs direct execution simulation, using a conservative time window approach. To overcome a small lookahead size resulting from switch level simulation, they use local barriers and predictive barrier scheduling. Local barriers use a nearest neighbor approach to reduce the number of nodes each processor must synchronize with. Predictive barriers reduce the number of required synchronization points by taking advantage of the fact that processors need not synchronize during periods when the simulated processors are not communicating. This is another example of increased lookahead, and works well when the simulated processes engage in long computational periods between communication. Both compile-time and runtime analysis are employed to predict when simulated processors are going to communicate.

LAPSE[4] is another simulator that performs direct execution of instrumented programs, in this case message passing programs. The granularity of synchronization is larger since there are larger periods of execution between message events.

Parallel Embra[10] is the simulator most closely resembling PPAint. It, too, executes on a shared memory platform. It differs from PPAint in using a largely direct-execution model, though mechanisms are provided to alter the model of most architectural features. Little is published about it; its synchronization mechanism seems to be a conservative time window approach with late messages simply being moved into the present.

7 Conclusions and Future Work

The development of PPAint has demonstrated that shared memory provides an effective substrate for distributed architectural simulation. Its most appealing characteristic is the ease with which synchronization can be implemented and refined. The efficiency of the model encourages experimentation with communication-intensive synchronization algorithms that would be impractical in a message-based system.

Numerous avenues for continued effort are apparent: evaluating scalability on larger systems, and on DSM systems in particular; determining which tasks have the largest effect on load imbalance and whether they can be made less "chunky;" evaluating the underlying shared memory performance, perhaps by simulating PPAint on top of PPAint, to gain insight into the dynamics of synchronization via shared variables.

References

[1] Brewer, E., Dellarocas, C., Colbrook, A., and Weihl, W. PROTEUS: A High-Performance Parallel Architecture Simulator. Tech. Rep. MIT/LCS/TR-516, Massachusetts Institute of Technology, Sept. 1991.

[2] Chandra, S., Larus, J. R., and Rogers, A. Where is Time Spend in Message Passing and Shared-Memory Programs? In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems* (Nov. 1994), pp. 61–75.

[3] Chandrasekaran, S., and Hill, M. D. Optimistic Simulation of Parallel Architectures Using Program Executables. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation* (May 1996), pp. 143–150.

[4] Dickens, P. M., Heidelberger, P., and Nicol, D. M. Parallelized network simulators for message-passing parallel programs. In *MASCOTS 95* (Jan. 1995).

[5] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[6] Stoller, L. B., and Swanson, M. R. PAINT: PA Instruction Set Interpreter. Tech. Rep. UUCS-96-009, University of Utah, March 1996.

[7] Swanson, M. R., Davis, A., and Parker, M. Efficient Communication Mechanisms for Cluster Based Parallel Computing. In *Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC 97)* (February 1997), vol. 1199 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–15.

[8] Swanson, M. R., and Stoller, L. B. Direct Deposit: A Basic User-Level Protocol for Carpet Clusters. Tech. Rep. UUCS-95-003, University of Utah, March 1995.

[9] Veenstra, J., and Fowler, R. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS 1994* (Durham, NC, Jan. 1994), pp. 201–207.

[10] Witchel, E., and Rosenblum, M. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the 1996 International Conference on Parallel Processing* (Aug. 1996), pp. 99–107.