Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

MANAGING EVENT TRACES FOR A WEB FRONT-END TO A PARALLEL SIMULATION

Boon Ping Gan
Li Liu
Zhengrong Ji

Gintic Institute of Manufacturing Technology
71 Nanyang Drive
638075 SINGAPORE

Stephen J. Turner
Wentong Cai

School of Computer Engineering
Nanyang Technological University
Nanyang Avenue
638798 SINGAPORE

ABSTRACT

To enhance the widespread use of a parallel supply chain simulator, a web front-end that enables access at any time and from any location has been developed. The front-end provides the capability of model uploading, simulation runs initiation, simulation activities visualization, and simulation statistics collection. Visualizing the simulation activities requires the parallel simulator to record event traces to the file system for displaying purposes. To minimize the negative impact of the recording on the performance of parallel simulation, an event trace management algorithm, coupled with a buffering mechanism, is proposed here. The approach was evaluated using six sample supply chain models. The results show that the proposed algorithm is capable of maintaining the same level of speedup when recording is performed (in the range of 2.5 to 3.0 on a 4-CPU shared memory system), as compared to runs without recording of event traces.

1 INTRODUCTION

Simulation has been used for supply chain management to simulate the flow of materials and information through multiple stages of manufacturing, transportation, and distribution (Jain et al. 1999). The simulation provides the capability of performing optimization on certain aspects of the chain, such as the inventory control policy. It can be used to study the impact of alternative control policies on the overall supply chain performance. To ensure a high confidence level for the decisions made based on the simulation, detailed modeling of each factory within the supply chain is required (Jain et al. 2000). This increases the complexity of the simulation, and in turn increases the execution time required to simulate the model.

In our work, parallel discrete event simulation has been used to improve the execution time of large supply chain models (Gan and Turner 2000). A conservative simulation protocol, an extension to an asynchronous protocol (Chandy and Misra 1979) for shared memory multiprocessor system, was used. Coupled with a dynamic load balancing algorithm, our parallel simulator has achieved a speedup factor of at least 2.5 relative to sequential simulation on a 4-processor shared memory system (Gan et al. 2000), simulating a semiconductor supply chain model (Jain et al. 1999) that is built from Sematech sample models (Sematech 1997). Simulating a large supply chain model is usually only feasible with some simplification to the model. With the capability of parallel simulation, such simplification is no longer necessary and the supply chain can be simulated in finer granularity. This offers a better confidence level for the decisions made.

Building a web front-end to interface to the parallel supply chain simulator is the focus of this paper. A web front-end is chosen as the medium of interfacing due to its capability of being available anywhere and at anytime. The interface allows users to a) upload their model, b) initiate simulation runs, c) visualize the simulation progress online/offline, and d) obtain the simulation statistics for analysis at the end of the simulation. The parallel execution of the simulation model is hidden from the users, and the interface appears like that of a conventional sequential simulation (Turner et al. 1998). The only distinction that possibly the users will notice is the speed of simulation. The parallel simulator completes the model simulation in a much shorter time as compared to the sequential simulator.

The critical aspect for the web front-end is its ability to visualize the progress of the simulation, either online/offline. This implies that the simulator needs to record event traces to the file system while the simulation is running. Recording of event traces will have a significant impact on the performance of simulators, particularly parallel simulators. The parallel simulation will be slowed down significantly if no extra care is taken to manage the mutual exclusive access to the file system. In this work, an efficient and effective algorithm is proposed for this purpose.

This paper is organized as follows: Section 2 describes the basic architecture of the web front-end to our parallel simulator, together with an overview of the design of the visualization support. This is then followed by a discussion on the algorithm that manages event recording in Section 3. Section 4 provides a performance evaluation of the algorithm to study the impact of the event recording on the parallel simulation. The paper is then concluded in Section 5 with some suggestions for future work.

2 THE WEB FRONT-END

2.1 Overview

As discussed earlier, the web front-end for the parallel simulator provides the following interfaces: a) to upload the supply chain model, b) to initiate simulation of the uploaded model, c) to visualize the progress of the model simulation, and d) to display the simulation statistics for analysis. Figure 1 shows the system architecture of this interface, which adopts a client-server architecture. The top level interface is displayed through an HTML page at the client side. The client will transmit the user selection to the web server. A PERL program is responsible for initiating the parallel simulator at the server. While the simulation is running (at the server), the parallel simulator records event traces to multiple files that are then used by a Java applet for display purposes. The graph shows the on going activities of each factory of the supply chain, and is updated as the simulation progresses. The trace files are sent to the Java applet (client side) through a Java program (server side) using a TCP/IP connection.

2.2 The Supply Chain Simulator Visualization Support

Visualization is an important tool in understanding a simulation. Having the ability to see what is happening during the simulation in general offers several benefits. Firstly, visualization is always an effective tool to explain the model to non-technical people, and people in higher management. Secondly, engineers can use visualization as a tool to verify the correctness of their model. This is always more effective than by just looking at the model statistics at the end of the simulation, from which it is difficult to draw any concrete conclusions on the model's correctness. Thirdly, visualization is useful in identifying a sudden but short interval of surging in some model variables, which is not easy to identify through average statistics collected at the end of the simulation.

The visualization support for the supply chain simulator takes a hierarchical form. The information is divided into three levels of hierarchy, namely i) the factory level, ii) the process flow level, and iii) the resource level. This logical arrangement derives naturally from the composition relationship among the entities of the domain model. Typically, a supply chain is made up of multiple factories that are interacting with one another. Within each factory, raw materials are transformed to finished goods by going through steps that are defined by the process flow. Within each step, certain resources, such as machine and operator, are needed for the processing. Thus, a hierarchical view is the most natural design for the visualization support.

Table 1 summarizes the information that is available for visualization at different levels of hierarchy. Each piece

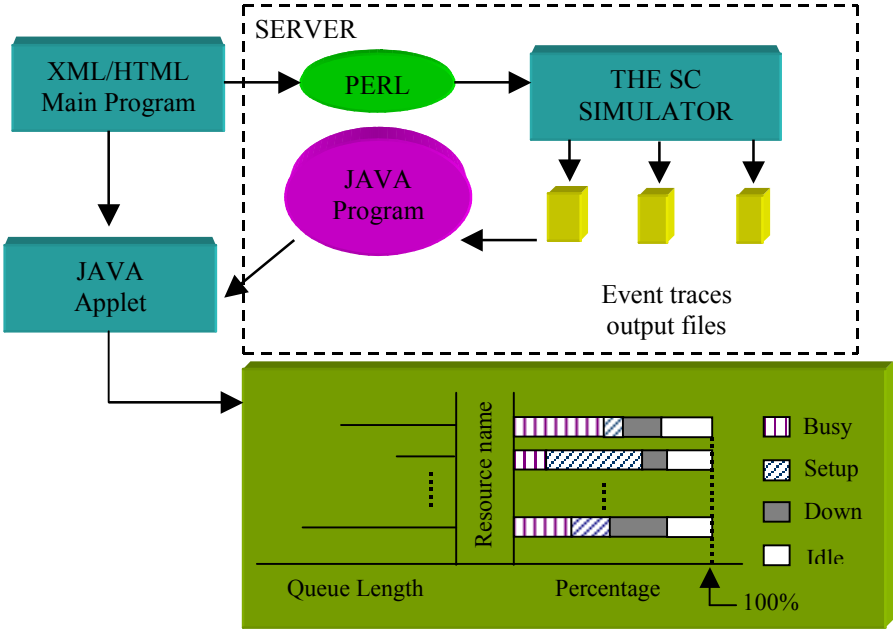

Figure 1: System Architecture

of information is updated at the Java front-end (Figure 1), as the simulation progresses. The trend of work-in-progress for each factory is plotted at the top level of the hierarchy (factory level). The second level (process flow level) displays the number of lots waiting at each step of the process flow. The information on the resources of the factory, such as resource queue length and percentage of time each resource spends on processing lots, changing setup, breakdown, and idle, is displayed at the bottom level (the resource level).

Table 1: Information Viewable at Different Levels of Hierarchy

Level	Information
Factory	Work-in-progress
Process flow	Number of lots at each step of process flow
Resource	Queue length at resources Resource busy percentage Resource setup percentage Resource down percentage Resource idle percentage

To display the information shown in Table 1, event traces for each simulation run are collected and recorded to the file system. The Java applet (Figure 1) receives the event traces from the Java program (server side) and translates them to information that is usable for display purposes. To avoid overflowing users with too much information, users are only allowed to look at the information of one hierarchy level at any one time. This helps them to focus on one aspect of the whole simulation model, and improves their effectiveness in analyzing the simulation. To facilitate this, event traces are segregated to multiple trace files. The segregation could be based on factory, process flow, or model resource for our supply chain model. Segregation by process flow and model resource is not feasible since there are hundreds of process flows and thousands of resources in a typical semiconductor supply chain model. Managing such a large number of files will be very inefficient. Thus, the most natural option to segregate event traces is by factory for our supply chain model. This design decision needs to be made before our event trace management algorithm is applied.

In addition to improving users' effectiveness in analyzing simulation runs, limiting the information being displayed also offers the benefit of transmitting a smaller amount of information from server to client. This helps to reduce the bandwidth requirement of the visualization support, which makes realizing it across the Internet feasible.

3 THE EVENT TRACE MANAGEMENT ALGORITHM

3.1 The Buffering Mechanism

Recording of event traces from a simulator to the file system needs to be done carefully to minimize its impact on the performance of the simulation. The general practice of ensuring minimum performance impact is to buffer event traces before writing to the trace files. Event traces will only be flushed when the buffer is full. This approach helps to reduce the frequency of I/O operations, which is the primary overhead of event recording. This reduction is even more crucial in parallel simulation when more than one process/thread might initiate an I/O operation to the same trace file at any one time. To avoid inconsistency in the state of the trace files, each file must only be accessed by one process/thread at any one time. This mutual exclusion of file access might waste CPU cycles if flushing of event traces is done too frequently (assuming the implementation uses a busy waiting approach). Thus, buffering offers an additional benefit of reducing the frequency of concurrent requests for file access in parallel simulation.

Figure 2 depicts the buffering mechanism of the proposed event trace management system. It assumes there are m trace files and n execution streams being created for each simulation run. Execution streams could be associated with entities of the simulation that can be executed in parallel, such as logical processes (LPs), or with threads. Even though this design is applied to the parallel supply chain simulator discussed in the earlier section, the general principles of this design can also be applied to other cases. The buffering mechanism was designed with two principles in mind: i) to reduce or eliminate the number of concurrent requests for access to shared resources such as files and buffers, and ii) to minimize the duration for which a shared resource is locked (locking is needed to avoid concurrent access to the same file). The first principle is realized by associating a buffer with each output trace file within each execution stream. By having one buffer per trace file, it eliminates the need for segregating the event traces when they are flushed (to file). In addition, having a set of these buffers for each execution stream minimizes concurrent requests for access (by execution streams) to the same buffer. No writing conflicts will ever occur during the writing process since each execution stream is writing to its own set of buffers.

Access conflict to the buffers will only arise when the contents of these buffers need to be flushed to the trace file. The first execution stream, denoted as T^F hereafter, that sees a full buffer initiates the flushing of all buffers associated with that trace file. This execution stream will compete for the buffers with the write access by other execution streams. This is unavoidable but its impact can be minimized by reducing the duration of buffer locking

(the second principle). A double buffering approach is employed to limit the locking duration on the swapping operation ($bufA_{i,j}$ and $bufB_{i,j}$ in Figure 2). Subsequently, T^F can perform the necessary operations on the swapped buffers, without interfering with the write access of other execution streams. The purpose of each buffer type, *bufA*, *bufB*, *saveBuf*, and *mergeBuf*, will be described together with the event trace management algorithm in the following section.

3.2 The Algorithm

Besides ensuring an efficient buffer management mechanism, the event trace management system also needs to ensure the correct ordering of events being written to the trace files. It is crucial to ensure the timestamp order of events since the web front-end is mirroring the simulation model activities, which occur in time order. To ensure the correct timestamp ordering in a sequential simulation is not difficult. However, there are some synchronization problems associated with parallel simulation, in which the order of events being written cannot be guaranteed since there are event traces from multiple execution streams. Hence, an algorithm that can ensure the correct ordering of events being written to the trace files is required for parallel simulation.

Figure 3 shows an event trace management algorithm that is capable of resolving the problem described above. This algorithm has to work hand-in-hand with the buffer management mechanism presented in Section 3.1. Each execution stream is allocated *m* of *bufA*, *bufB*, and *saveBuf* buffers (refer to Figure 2). Execution stream, *i*, writes event traces to its j^{th} *bufA* buffer, represented as $bufA_{i,j}$, after each event for that trace file is simulated (lines 3-6). When the execution stream detects that the $bufA_{i,j}$ buffer reaches the full state (line 7), it will initiate the flushing operation on this buffer and all the j^{th} *bufA* in other execution streams. It is important to note that any

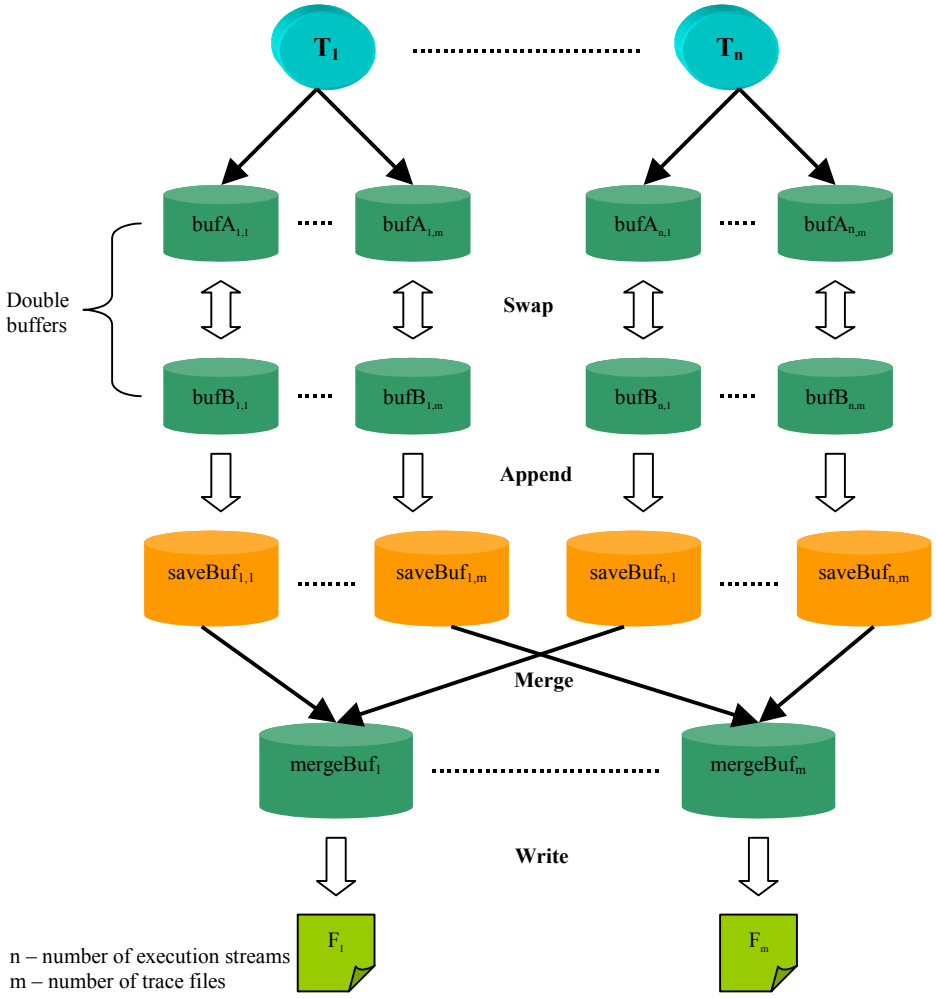

Figure 2: Buffering Mechanism for Event Recording

execution stream can initiate the flushing. This leads to a possibility of execution streams contending among each other to initiate the flushing operation (of the j^{th} buffer). A *mergeLock* is thus associated with each of the *m* buffer sets (line 8 and 22) to avoid concurrent flushing on the same buffer set.

```
1.  Let e be an event that is simulated at
      execution stream, Ti, where i ∈ {1,2,…,n}
2.  simulate(e)
3.  j = e.get_trace_file_id()
4.  bufAi,j.lock()
5.  bufAi,j.push(e)
6.  bufAi,j.unlock()
7.  if (bufAi,j.is_full()) then
8.   mergeLockj.lock()
9.   if (bufAi,j.is_full()) then
10.   safetimei = min(simulTimek) forall k= 1 to n
11. -- swap all the buffers of trace file j --
    --              for merging              --
12.   for all k = 1 to n do
13.     bufAk,j.lock()
14.     swap(bufAk,j, bufBk,j)
15.     bufAk,j.unlock()
16.     saveBufk,j.append(bufBk,j)
17.   endfor
18.   -- merge all sorted buffers of --
      --        trace file j          --
19.   mergeBufj = sorted_merge(saveBufk,j)
        forall k = 1 to n, where  e ∈ saveBufk,j,
        and e.timeStamp < safetimei
20.   mergeBufj.flush_to_file(j)
21.  endif
22.  mergeLockj.unlock()
23. endif
```

Figure 3: The Event Trace Management Algorithm

Associating a *mergeLock* with the buffer sets can indeed avoid concurrent flushing of the same buffer set, but there is still the possibility of flushing an empty buffer. This happens when more than one execution stream detects that a buffer set is full at the same time. The first execution stream that successfully acquires the lock (*mergeLock*) will flush the content of the buffer set and releases the lock. Execution streams that acquire the lock next would have nothing to flush. To avoid this from happening, a second check on the buffer status is performed before the flushing operation is initiated (line 9). This eliminates the problem described.

When the flushing operation is initiated, the j^{th} buffer of the corresponding *bufA* and *bufB* buffers for all the execution streams are first swapped (swapping of *bufA* and *bufB* is implemented as a pointer swap instead of content copying) (line 13-15). This suggests a possibility of contention between the writing and swapping operations to *bufA*. A lock is thus associated with these operations. Upon swapping, the content of $bufB_{i,j}$ is appended to the content of the corresponding $saveBuf_{i,j}$ buffer (line 16). The *saveBuf* buffer is needed since not all event traces are flushed to the trace file when the flushing operation is initiated. Some event traces that are not yet safe to be flushed are kept in the *saveBuf* for the next initiation of the flushing operation. An event is considered as not safe when there is a possibility that a subsequent event that is flushed to the file can happen earlier (in terms of simulation time) than itself. To avoid this from happening, a safetime needs to be computed. It is assigned the global minimum simulation time of all execution streams (line 10). Thus, all events with timestamp less than this value are in fact safe to be flushed.

Before the safe events within all the j^{th} *saveBuf* can be written to the trace file, these events need to be sorted first (line 19). An efficient merging algorithm, that keeps items from multiple lists sorted as the items are merged, is employed for this purpose. The content of all the j^{th} *saveBuf* are merged to the corresponding *mergeBuf*. Upon completion, the content of the *mergeBuf* will then be flushed to the j^{th} trace file (line 20). This will complete the flushing operation and the execution stream will go back to its normal operation, which is the simulation of events and pushing events to the *bufA* buffer.

3.3 Applying the Algorithm to the Parallel Simulator

The parallel simulator is capable of simulating a semiconductor supply chain model (Jain et al. 1999) that is defined using the Sematech data format (Sematech 1997). A resource view is adopted by the parallel simulator, whereby resources, such as machines and operators of the model are mapped to logical processes (LPs) of the parallel simulation (Turner et al. 1998). But having a 1-to-1 mapping of resources to LPs will be too fine grain to achieve good parallel performance. Thus, a many-to-1 mapping is adopted to increase the LP granularity (Gan et al. 2000). These LPs are put into a global pool at the beginning of the simulation. Threads that are idle will get an LP from the pool and simulate the LP. If the LP cannot make simulation time progress, the LP will be returned to the global pool. Each LP can thus be executed by different threads at different times. This helps to dynamically balance the system workload and improve the parallel performance.

To apply the event trace management algorithm to the parallel simulator, two questions need to be answered. Firstly, what should be mapped to the execution stream, whether LPs or threads are the better option? Secondly, what is the right buffer size for all the *bufA* (and *bufB*) to cause initiation of the event flushing operation? For the first question, mapping LPs to the execution stream will be the most logical and efficient approach following the decision on event trace segregation discussed earlier. The other option of mapping threads to execution streams introduces an additional requirement of explicitly sorting the content of *bufA*. If no sorting is done, the events will not be in order and this invalidates the algorithm shown in Figure 3. It would be possible for out of order event pushing to occur (by different LPs) since LPs that are executed by the same

thread are not necessarily at the same simulation time at any instant of the parallel simulation. This extra overhead of sorting *bufA* is not necessary with the mapping of LPs to execution streams.

The answer to the second question is not as obvious as the first. It has to be answered by performing some experiments to determine the most appropriate buffer size to be used, as described in Section 4. A small buffer will increase the frequency of initiating the flushing. This in turn increases the frequency of buffer contention among the execution streams. By increasing the buffer size, the initiation frequency can be reduced. This reduction can translate to an improvement in execution time. But it is highly likely that the improvement becomes stagnant when the buffer size grows beyond a certain threshold value. These arguments will be verified through experiments described in Section 4.

4 PERFORMANCE STUDY

This section describes two experiments that were performed. The first experiment was to vary the size of the *bufA* buffer and study its impact on the execution time of the parallel simulator. This experiment also verifies the advantage of using a buffering mechanism, as compared to the case in which no buffer is used for event tracing. Upon finding a suitable buffer size, the second experiment was conducted to verify if the same level of speedup is maintainable, by comparing runs with event tracing to runs without event tracing. The experiments were performed on a Sun Enterprise 3000 system, that has four 250 MHz UltraSparc II processors and 512 Mbytes of memory. The parallel simulator was implemented using C++ and compiled with GNU GCC compiler version 2.95.1. The supply chain models being used for the experiments were constructed from six sample models, based on Sematech sample models and models from past industry projects. The simulation run length was set to 100 days.

Figure 4 plots the execution time achieved as the buffer size was varied from 0 bytes to 102400 bytes. Event traces are not sorted in timestamp order for scenarios that use 0 bytes buffer size. Subsequent sorting is required for these cases but the sorting time involved is not included in the figure. Even so, it is obvious from the figure that the execution time achieved with scenarios that use buffers is already consistently better than those that use no buffer (0 buffer size). It is indeed beneficial to introduce buffers, as a temporary storage to hold event traces, between the simulation program and the trace files. Another crucial observation is that the execution time improves as the buffer size is increased. From the experiments, it seems that a buffer size of 10240 bytes is sufficient to ensure good performance for the parallel simulation, although this improvement is diminishing as the buffer size grows beyond 1024 bytes.

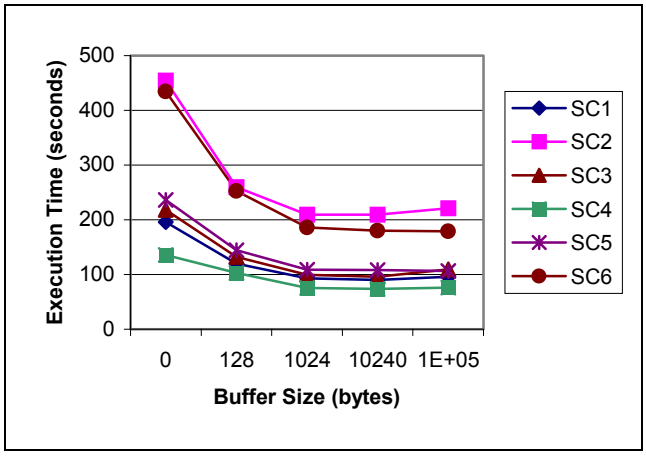

Figure 4: Impact of Buffer Size on Execution Time

Figure 5 compares the speedup achieved for runs with event-tracing and runs without event-tracing. Each set of these runs is performed for both sequential and parallel simulation. For the runs that output event traces, buffers are also introduced to the sequential simulator to guarantee a fair comparison. The speedup is computed based on the corresponding runs, that is, sequential and parallel runs with event tracing, and sequential and parallel runs without event tracing. As can be seen, the speedup achieved is maintained at the same level when the event trace management algorithm is used. This shows that the algorithm is effective in minimizing its impact on the performance of the parallel simulation, which is crucial since file I/O operations have always been a major performance bottleneck.

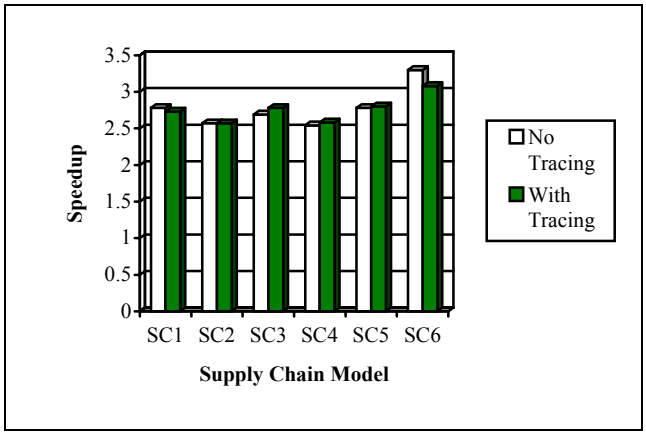

Figure 5: Speedup Achieved With and Without Event Tracing

5 CONCLUSIONS

This paper has presented a web front-end together with an effective event trace management algorithm for a parallel simulator. Through the web front-end, the parallel simulator can be accessed from anywhere at anytime using any

web browser. Further, this front-end presents an interface to the parallel simulator which is like that of a conventional sequential simulation. The interface allows the user to submit their model, initiate simulation, visualize the simulation, and collect the simulation statistics. The visualization facility requires the parallel simulator to record event traces to the file system. This leads to the development of an effective event trace management algorithm that minimizes the impact of event recording on the performance of the simulation. It is proven that the algorithm is capable of maintaining the same level of speedup even when file I/O is required to record the event traces. Future work will focus on realizing interactive control of the simulation runs to allow users to perform *what-if* analysis while the simulation is running.

ACKNOWLEDGMENTS

This research is supported by National Science and Technology Board, Singapore, under the project: Parallel And Distributed Simulation of Virtual Factory Implementation. It is a collaborative project between Gintic Institute of Manufacturing Technology, Singapore and the School of Computer Engineering in Nanyang Technological University, Singapore. The project is located at the Centre for Advanced Information Systems at Nanyang Technological University, Singapore.

REFERENCES

Chandy, K.M., and J. Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering SE-5*: 440-452.

Gan, B.P., and S.J. Turner. 2000. An asynchronous protocol for virtual factory simulation on shared memory multiprocessor systems. *Journal of Operational Research Society Special Issue on Progress in Simulation Research* 51: 413-422.

Gan, B.P., Y.H. Low, S. Jain, S.J. Turner, W. Cai, W.J. Hsu, and S.Y. Huang. 2000. Load Balancing for Conservative Simulation on Shared Memory Multiprocessor Systems. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, 139-146.

Jain, S., B.P. Gan, C.C. Lim, and Y.H. Low. 2000. Bottleneck Based Modeling of Semiconductor Supply Chain. In *Proceedings of the International Conference on Modeling and Analysis of Semiconductor Manufacturing*.

Jain, S., C.C. Lim, B.P. Gan, and Y.H. Low. 1999. Criticality of detailed modeling in semiconductor supply chain simulation. In *Proceedings of the 1999 Winter Simulation Conference*, ed. P.A. Farrington, H.B. Nembhard, D.T. Sturrock, and G.W. Evans, 888-896. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Sematech. 1997. *Modeling Data Standards, Version 1.0*. Technical Report, Sematech Inc., Austin, TX 78741.

Turner, S.J., C.C. Lim, Y.H. Low, W. Cai, W.J. Hsu, and S.Y. Huang. 1998. A Methodology for Automating the Parallelization of Manufacturing Simulations. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, 126-133.

AUTHOR BIOGRAPHIES

BOON PING GAN is an Associate Research Fellow with the Manufacturing Planning & Scheduling group at Gintic Institute of Manufacturing Technology, Singapore. He received a Bachelor of Applied Science in Computer Engineering and Master of Applied Science from Nanyang Technological University of Singapore in 1995 and 1998 respectively. His research interests are parallel and distributed simulation, parallel programs scheduling, and application of genetic algorithms. His email address is <bpooi@gintic.gov.sg>.

LI LIU is an Associate Research Fellow with the Manufacturing Planning & Scheduling group at Gintic Institute of Manufacturing Technology, Singapore. She received her Bachelor of Science in Mathematics from Beijing University in 1991 and Master of Computer Engineering from Academic Sinica, China in 1994. Her research interests are parallel discrete event simulation, supply chain web-based simulation. Her email address is <lliu@gintic.gov.sg>.

ZHENGRONG JI is a Research Associate with the Manufacturing Planning & Scheduling group at Gintic Institute of Manufacturing Technology, Singapore. He received his Bachelor of Applied Science in Computer Engineering from Nanyang Technological University of Singapore in 1999. His research interests are parallel and distributed simulation, distributed algorithm and mobile computing. His email address is <zrji@gintic.gov.sg>.

STEPHEN J. TURNER is an Associate Professor in the School of Computer Engineering at Nanyang Technological University (Singapore). Previously, he was a Senior Lecturer in Computer Science and Director of the Parallel Systems Research Laboratory at Exeter University (UK). He received his MA in Mathematics and Computer Science from Cambridge University (UK) and his MSc and PhD in Computer Science from Manchester University (UK). His current research interests include: distributed and parallel simulation, visual programming environments, parallel algorithms and languages, distributed computing and agent technology. His email and web addresses are

<assjturner@ntu.edu.sg> and <www.ntu.edu.sg/home/assjturner>.

WENTONG CAI is an Associate Professor in the School of Computer Engineering (SCE), Nanyang Technological University (NTU). He obtained his B.Sc. in Computer Science from Nankai University (P.R. China) in 1985, and Ph.D. from Exeter University (UK), also in Computer Science, in 1991. Since 1990, he has been actively involved in the research in the areas of Parallel and Distributed Simulation (PADS). Dr. Cai is a member of IEEE and his current research interests include: PADS, Parallel Programming Tools and Environments, and Performance Analysis. His email and web addresses are <aswtcai@ntu.edu.sg> and <www.ntu.edu.sg/home/aswtcai>.