EFFICIENT LARGE-SCALE PROCESS-ORIENTED PARALLEL SIMULATIONS

Kalyan S. Perumalla
Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, Georgia 30332-0280, U.S.A.

ABSTRACT

Process oriented views are widely recognized as very useful for modeling, but difficult to implement efficiently in a simulation system, as compared to event oriented views. In particular, the complexity and runtime overheads of the implementation have prevented the widespread use of process oriented views in optimistic parallel simulations. Here, we review the conventional approaches to implementing process-oriented views, and outline some of the sources of problems in those approaches. We also identify an approach that we call *stack reconstruction*, which is most suited for portably and efficiently supporting optimistic process-oriented views. Benchmark simulations using our preliminary implementation, which is incorporated in the TeD modeling and simulation system, confirms the low overheads of this approach, and demonstrates its capability to simulate over one million processes in a process-oriented model.

1 INTRODUCTION

Three widely recognized world views for simulation are: event-oriented view, process-oriented view and activity-scanning view (Mitrani 1982). The first two views are the more widely used among the three views. All the three views are equivalent in the sense that process-oriented and activity-scanning views can be translated into semantically equivalent event-oriented views. An advantage of using the process-oriented view is that models tend to be smaller, and easier to develop and understand, making it more appealing to the modelers. On the other hand, it is generally perceived that event-oriented views can be more efficiently implemented than the other two views, especially in the context of optimistic parallel simulations.

In recent modeling and simulation efforts, there is a clear demand for the capability to support very large scale simulation. The need for *parallel* simulation is clear in these application domains. In addition, the models tend to be too complex to express using the low-level primitives of the event-oriented view, thus making process-oriented view a natural choice for developing such large and complex systems. This makes it important to find ways to efficiently support very large number of processes in a process-oriented modeling and simulation system.

Specifically, our experiences with modeling large and complex telecommunication networks using the TeD language (Perumalla et al. 1998, SIGMETRICS PER 1998) served as our initial motivation to find efficient implementation alternatives for process-orientation. For example, in many interesting configurations (such as global Internet) of network models in TeD, the number of processes easily exceeds one million, warranting efficient support for large-scale process orientation. Also, modelers demand support for full process orientation, such as the ability to invoke `wait` statements over nested procedure calls, at almost arbitrary points in a procedure body. Since TeD permits the direct embedding of C^{++} code in the models (see Perumalla et al. 1998), we are further constrained from exploring alternatives that trade-off modeling power for efficiency.

The interaction of parallel simulation techniques with the implementation of large-scale process-oriented views (supporting millions of processes per simulation) generates challenges, such as minimizing the memory size and memory copy requirements of the implementation. Process-oriented views are generally perceived to be expensive in the context of optimistic parallel simulations. This is mainly due to the fact that processes entail state-saving overheads for the maintenance of additional control flow information and transient data. These overheads can be quite large in naive implementations, unless they are carefully reduced to the minimum necessary.

Related Work

A number of implementations of process-oriented views (which are mostly language or preprocessor-based) have been reported in the recent years. The Maisie language (Bagrodia and Liao 1994) supports the concept of process in the form of an entity description, along with support for the optimistic parallel simulation of Maisie entities. However, Maisie does not include direct support for the suspension of a stack of nested procedure calls (wait statements cannot be used inside functions invoked by entities). The macros-based approach of IMPORT/SPEEDES (Whitehurst and Brutocao 1998) also seems to limit simulation time advances to the main process body. Apostle (Booth and Bruce 1996) is a new language that implements process-orientation using continuations, with support for optimistic parallel simulations. Apostle, however, has specialized semantics, which do not carry forward well to our domain of interest, which is C^{++}. Other languages and packages, such as MODSIM and Parasol support process-orientation views, but without great success in optimistic simulations. More recent work includes the Nops system (Poplawski 1998) which reports low overhead implementation of processes. Nops, however, only supports *conservative* parallel simulation, and it has no direct support for advancing simulation time over a stack of nested procedure calls.

Here, we identify an approach that we call *stack reconstruction* as most suited for portably and efficiently implementing optimistic parallel simulations of "true" process oriented views in an expressive language such as C^{++}. First, we define what constitutes true process orientation, in section 2. Next, we briefly review some conventional implementation approaches in section 3, identifying their problems in the context of optimistic parallel simulations. We then describe the stack reconstruction approach with details of implementation, followed by some salient performance results which indicate the low overheads of the approach.

2 PROCESS ORIENTATION

A process is a distinct flow of control, containing a combination of computation and synchronization operations. Processes typically synchronize with each other by exchanging events. Process orientation is an elegant way of capturing context information under the conventional procedural programming paradigm. (Process or event orientation is orthogonal to object orientation. In an object-oriented setting, procedures in fact correspond to method calls.) The main body of a process can invoke procedures which in turn can invoke other procedures.

Table 1: Features of an Ideal Process-oriented System

F1	Procedures can declare and use local variables
F2	Procedure calls can be nested
F3	Procedures can be recursive and re-entrant

(a) Programming style

F4	Primitives to advance simulation time can be invoked in any procedure
F5	Primitives to advance simulation time can be invoked wherever a conditional, looping or other statements can appear.

(b) Time control

Functionality

The ideal modeling capabilities of a process are listed in tables 2 (a) and (b). Note that the features are generally orthogonal to each other (i.e. it is possible to pick and choose a subset of the features that will be adopted by a process-oriented system). A *pure* process-oriented view is one in which all the features **F1** through **F5** are supported. The first three features are those that are expected of any modern languages supporting the procedural programming style, and expected by most modern programmers. The last two features are specific to the simulation domain — time advancing primitives are those that serve to advance simulation time, such as `wait`, or `hold` statements and other such variants. It is the interaction of the programming style with the simulation time advances that gives rise to interesting challenges in implementing process-oriented views efficiently.

Process Type Continuum

Although pure process-oriented views are useful in some complex models, more restrictive definitions of processes can be made for use in some models which do not warrant the full power of pure process-orientation.

By varying the combination of features that are supported, we can achieve variation in the efficiency of implementation. For example, if the features **F4** and **F5** are unused by a process-oriented model (i.e., simulation time is advanced only at the top–level statements in the main process body, as opposed to under **if** statements or inside a procedure), then such a model can be implemented in a way that incurs no more overheads than if that model was re-written using event-oriented view. In fact, such processes are nothing but event-oriented models expressed in a way that better exposes context information. Such models with lower demands on the expressive power do appear in real-life modeling, such as cited in Perumalla et al. (1997).

Similarly, if we relax the feature **F1**, and enforce the rule that all local variables are *immutable* (i.e., never change after initialization), or if local variables are not supported at all by the modeling language, then, issues such as re-entrancy and recursion become easier to handle in the implementation.

Other simplifications (such as in Maisie, IMPORT and Nops) preclude the feature **F4**, but do support feature **F5**. In other words, although simulation time cannot be advanced in procedures, it can be advanced at any point in the process body (say, under conditional and looping statements). In such implementations, a stack of suspended procedures must be indirectly *emulated* using a chain of stack-less processes. The issues of local variables and recursion are also simpler to handle, especially in optimistic simulations, due to the reduction in the amount of information to state-save, and, the issues of local variables and re-entrancy can be effectively delegated to the host language, such as C^{++}.

3 IMPLEMENTATION ALTERNATIVES

There exist several techniques for implementing process-oriented views, of which we outline the important ones. One approach is to view the control and data information of a process as a black box, and preserve its contents across suspension and resumption points — this is the *threads*-based approach, such as in Mascarenhas and Rego 1996. Another approach is to define the modeling language semantics in such a way as to effectively remove the need for a stack — the *continuations*-based approach of Booth and Bruce 1996 is an example. Yet another approach is to transparently maintain auxiliary information that is barely sufficient to restore the native stack of a suspended process — which is the *stack reconstruction* approach described here. We describe each of these approaches next.

Process Stack

Many modern languages (compilers, to be precise) use an optimization, called *physical* or *native stack*, for the very frequent type of operation: procedure call. A native stack is an encoding of program counter, return addresses, local variables and argument lists (Koopman). Although the logical stack can be implemented in other ways (say, using linked lists), it is very often represented in contiguous memory locations for performance reasons. The native stack is used to efficiently implement the procedure call semantics by pushing and popping invocation information on and off the stack. Since the memory size of local variables and argument lists varies across procedures, the native stack serves to optimize the memory allocation and deallocation operations, by exploiting the contiguous memory feature of the native stack.

The preceding way in which native stacks are used is tightly coupled to the operation of executable code that most compilers generate, and tightly integrated into the way many operating system services (such as signals) operate.

Since the procedure stack is precisely what is necessary to support process-oriented views, it is natural to attempt to utilize native stacks to implement the simulation processes. This is the approach taken in implementations based on *threads*.

Threads

Threads are light-weight computation abstractions widely used in many domains such as high performance computing and multi-media servers. Threads provide support for multiple process stacks which can communicate and synchronize with each other. Several multi-threading packages exist that provide operating system-level threads and/or user process-level threads (Tanenbaum 1995). Portable implementations of threads are available, which typically make use of standard facilities such as the POSIX calls `setjmp()` and `longjmp()`. Threads can be scheduled, suspended and resumed. Each thread typically contains a stack of procedure activation frames, although some optimized packages use special thread synchronization semantics to avoid using physically distinct stacks for each thread.

One way to implement simulation processes is to use a single thread for each process. The thread suspension and resumption primitives can be used to achieve the simulation time advances in the simulation process code. Thus, for example, a `wait` statement in the simulation process will be mapped to suspending the thread of the simulation process, and handing control over to the simulator's scheduler. At the instant the waiting condition is satisfied, the process is resumed just after the wait statement, by resuming the thread of the simulation process.

The advantage of using a threads to implement simulation processes is that little additional implementation work is necessary to save and restore the stacks of the simulation processes. The simulator only acts as the thread scheduler. There are, however, several drawbacks of threads-based implementation. Either thread stacks bump into each other (thread stacks typically do not grow), or the memory requirements can be high to support very large number of processes. Thread migration is either unsupported or expensive. More importantly, conventional threads are difficult to optimize for optimistic parallel simulation (as discussed in more detail in the next section). The design of off-the-shelf thread systems may not be well suited to scale to millions of active threads. Special large-scale multi-threading systems exist, which could potentially be useful in sequential and conservative parallel simulations; but they have not been tested for use in optimistic parallel simulation.

Fundamentally, conventional threads are general-purpose computation abstractions, with potentially complex inter-thread synchronization, and scheduling disciplines. Simulation processes, however, have simple and well defined suspension and resumption semantics (based on simulation time advances), and a simple scheduling discipline (usually, least timestamp first). Whereas support for preemptive threads incurs overheads such as saving register state, simulation processes on the other hand need never incur such overheads, due to their simpler scheduling discipline.

Continuations

Continuations (Appel 1992) constitute another efficient mechanism for implementing processes. Suppose we define the modeling language in a way that allows the compiler to cast all language constructs (such as conditional or looping statements) into separate blocks of non-interruptible operations. In such a case, the process can be implemented as a special form of a finite state machine, in which each state dynamically designates its successor, called a *continuation*. If we ignore the issue of local variables for simplicity, it is clear that the process context is fully represented just by the identity of the current continuation (pointer to a function) of the process. Further optimizations are possible whereby explicit storage of the per-process continuation information can be avoided, and implicitly recorded on the run-time stack of the simulator (Booth and Bruce 1996).

To effectively implement this technique, either special language constructs have to be defined, or compilers of existing languages have to be modified (Appel 1992).

4 STACK RECONSTRUCTION

Another approach to implementing process-oriented views is what we call *stack reconstruction*. The underlying idea is that, instead of saving and restoring the contents of the native stack, we separately maintain information at runtime such that the native stack can be reconstructed to the same state in which it was when the process was suspended. This not only allows us to throw away the unnecessary contents of the native stack, but also permits us to easily capture modifications to the process state, which is essential for state-saving operations in optimistic parallel simulation.

We use a compiler-based solution for supporting this approach transparently, leaving the models uncluttered with the implementation details. We assume the models are translated into some general purpose programming language code, such as C^{++} which is then compiled to result in executable models. We use C^{++} as the target language in our examples.

```
procedure one()
{
    ...
    wait(c2)
    ...
}
procedure two()
{
    s1
    if(...) {
        s2
        wait(c1)
        s3
    }
    for(...) {
        s4
        call one()
        s5
    }
}
process p()
{
    ...
    call two()
    ...
}
```

Figure 1: Model Code of a Process `p` which Invokes `two()`, which in turn Waits on a Condition, and Invokes `one()`

To understand how the stack reconstruction approach works, consider the pseudocode fragment in figure 1 of a process-oriented model containing two procedures `one()` and `two()`. For simplicity, we postpone the treatment of local variables and procedure arguments to later in the discussion. The procedure `two()` contains some (arbitrary computation) statements, `s1`, `s2`, $\cdots$, `s5`. In addition, it contains a simulation advance (`wait`) statement inside a conditional (`if`) statement, and a procedure call to `one()` inside a looping (`for`) statement.

Consider the execution of process `p` when it invokes the procedure `two()`. If and when the procedure execution reaches line 12, it must be suspended at the `wait` statement, and when the wait condition is satisfied, it must be resumed at line 13 with the statement `s3`. Similarly, when the execution reaches line 17, the procedure `one()` must be invoked; again, process execution may have to be suspended if the procedure `one()` invokes some other `wait` statement, and resumed at the correct position in procedure `one()` when the process is resumed.

```
 1.  int two()
 2.  {
 3.      switch(JI) {
 4.          case 0: goto start;
 5.          case 1: goto lbl_1;
 6.          case 2: goto lbl_2;
 7.      }
 8.      start: /*continue*/;
 9.      s1
10.      if( ... ) {
11.          s2
12.          wakeup = c1;
13.          JI = 1;
14.          return SUSPENDED;
15.          lbl_1: /*continue*/;
16.          s3
17.      }
18.      for(...) {
19.          s4
20.          lbl_2: flag = one();
21.          if( flag == SUSPENDED )
22.          {
23.              JI = 2;
24.              return SUSPENDED;
25.          }
26.          s5
27.      }
28.      JI = 0;
29.      return DONE;
30.  }
```

Figure 2: Extracts of Code Generated for `two()`

In the stack reconstruction approach, the compiler identifies and marks all the positions in a procedure at which a process suspension can occur. Lines 12 and line 17 qualify as suspension points of procedure `two()`. The compiler then assigns ordinal numbers 1 and 2 to the two suspension points. When the execution of procedure `two()` is suspended, it is sufficient to remember the ordinal number (or, jump index, JI) of the point where it was suspended. Using this number, we can directly jump (using a combination of `switch()` and `goto` statements at the beginning of the procedure) to the point where the execution was left off. This is demonstrated in figure 2 which lists the code generated from the model of figure 1. Note that the resumption point for a `wait` statement is just beyond the `wait` statement, whereas the resumption point for a procedure `call` statement is exactly at the same procedure `call` statement, which results in a re-invocation of the procedures.

The unrolling of native stack occurs when the process is suspended — all the procedures actually perform a `return`, returning control to the simulation system. The reconstruction of native stack occurs when the process is resumed — the procedure call chain is correctly reconstructed using function re-invocation, with the help of the saved ordinal numbers.

If each procedure can have at most 256 suspension points in its body (which is a reasonable limit for human-written models), a single byte is sufficient to record the jump index for each procedure. An array of jump indices can be used for each suspended process to record the jump indices of its active procedures. (The jump indices represent *forward* addresses albeit, more memory efficient, as opposed to the *return* addresses of conventional native stacks.)

Local Variables

Now let us consider the implementation of local variables. We maintain a pointer to a memory buffer (*frame*) along with each jump index. References to the local variables in the procedure body are translated to indirect (pointer) references to the frame. In optimistic parallel simulations, the frame remains allocated until the global simulation time (GVT) sufficiently advances to guarantee that the frame deallocation will not be rolled back.

Procedure arguments can be viewed as special type of local variables, which are initialized automatically by the compiler based on the procedure invocation. Hence, the compiler can treat them as such, and follow the same techniques as for local variables to save modifications to the arguments. A value of 0 for the jump index can be used by the compiler to distinguish between invocation and reconstruction, to enable it to initialize the arguments upon invocation of a procedure, and skip the initialization if the procedure is re-invoked during stack reconstruction.

Optimistic Simulation

The important feature of stack reconstruction that helps in optimistic simulations is that it allows to easily and transparently trap modifications to the logical stack. The logical stack consists of jump indices and local variables. Jump indices are incrementally state saved by the compiler, since the compiler manages them itself. Local variables can also be incrementally state saved using transparent incremental state saving techniques, such as those using overloaded assignment operators (Ronngren et al. 1996), by trapping modifications to local variables in the procedure body. The stack reconstruction approach is also appealing for optimistic parallel simulation due to its reduction in the amount of state-saved information.

In native stacks, it is the case that not only more extensive information is stored on the stack, but also it is hard to gain precise access to any and all modifications to that information. These factors together preclude efficient state-saving of modified information in optimistic simulations. Thus, if native stacks are used to implement process-oriented views in optimistic simulations, it is becomes unavoidable to perform a brute-force blind copy of the whole native stack for state-saving, resulting in large and expensive state-saving costs for every process context-switch. This has been the fundamental reason for the perception that process-orientation is expensive to support under optimistic parallel simulations. It is now clear that with a minimal intermediate translator, which maintains jump indices and local variables, the state-saving costs can be reduced to the minimum required.

This approach also allows for lazy and tight memory allocation, which is important when simulating very large number of processes. In our implementation, we perform lazy allocation of memory — memory for the frame is not allocated until the moment the frame is actually required, thus resulting in tight memory utilization, as opposed to preset stack limits of native stacks.

Another advantage of the stack reconstruction approach is its ability to support efficient run-time migration of active processes across heterogeneous platforms. Since machine-independent formats are used to represent the logical stack of procedure calls, it is both easy and efficient to pack a process stack, and move and restore it on the destination machine, even if the source and the destination machines use incompatible native stack representations. A further advantage is that the stack reconstruction implementation is completely portable, independent of the operating system or the native language compiler formats. It also has the desirable feature of not interfering with other compiler optimizations (such as tail recursion and register allocation).

Related Schemes

Other parallel simulation systems do implement variations on the scheme of using `goto`s (Maisie's code generator, IMPORT's macros, Cilk's frame-allocator), but do not allow nested procedure calls. Recent work on fault-tolerance systems (Ramkumar and Strumpen 1997) also utilizes another variant of this scheme to "walk up and down the stack" for portably checkpointing the C-runtime stack.

5 PERFORMANCE

We have incorporated the stack reconstruction technique into the implementation of true process-oriented views in the TeD modeling and simulation system (Perumalla et al. 1998). To test its capabilities and run-time performance, we have used three different scenarios: (1) to measure its process context switching costs in pure process-oriented models (2) to compare its performance against event-oriented models (3) to stress-test the approach with respect to size, using models containing over a million processes.

Context Switching Cost

The *active* depth of a stack is the depth of the stack (number of procedures on the stack) at the moment the process is suspended. When the stack reconstruction approach is used, it is clear that the cost of suspending or restoring a process is a function of the active depth of the restored (or suspended) stack, since the active functions are re-invoked (or unrolled) to restore (or free up) the stack. Figure 3 shows the

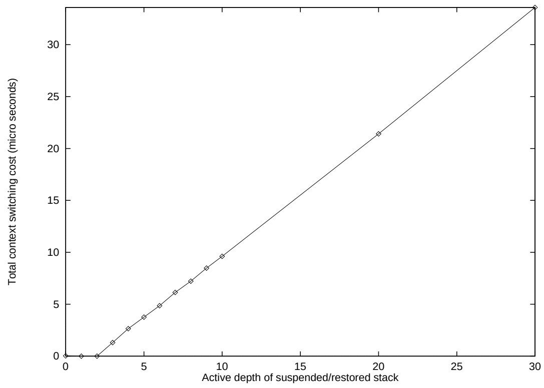


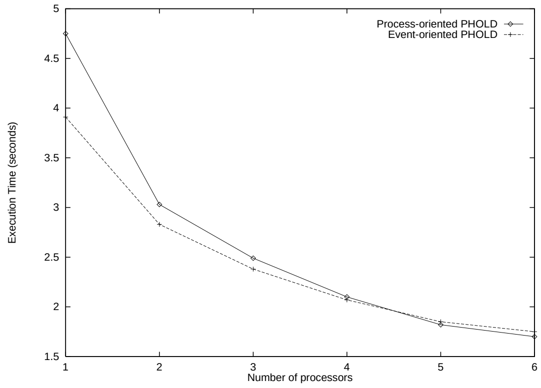
Figure 3: Variation of Process Context-switching Cost with Active Depth of Stack

variation of the context switching cost with the active depth of the stack, measured using a synthetic model containing processes having exactly same active depth of stack. The switching cost includes the costs of suspending a process and resuming another process, and also the cost of building and maintaining runtime information to reconstruct the stack of process. The benchmarks were run on an SGI Origin multiprocessor with `R10000` processors.

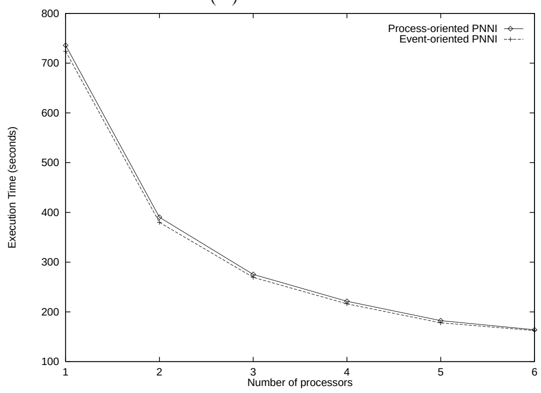
From the figure, it is seen that the context switching takes less than $10\mu s$ for processes with relatively small active depths of stack. In our experience with modeling large and complex telecommunication network protocols, the processes in our models never exceeded an active stack depth of 5, giving a context switching time of less than $4\mu s$. Figure 3 also serves as a means of locating the tradeoff points when the stack reconstruction approach performs better or worse than other alternative approaches, such as using off-the-shelf thread packages.

Comparison with Event Orientation

On certain simpler types of process-oriented models, the run-time performance of our implementation achieves the superior performance of equivalent event-oriented models. This is demonstrated in figures 4 (a) and (b), which plot the elapsed time of our process-oriented implementation versus that of equivalent event-oriented expression of the same models. We used one synthetic and one real-life model in this scenario. The figures correspond to the optimistic parallel simulation of the models, using Time Warp. The synthetic model is the well-known PHOLD application containing processes that exchange events in a way that conserves the total number of events in the model. The second model is the PNNI telecommunication network application (Perumalla et al. 1997) containing network models capturing the ATM Forum's standard on internetwork protocols called the PNNI (Private Network to Network Interface). Both the models share the feature that the simulation advance primitives (**wait** statements) appear only at the outer-most level in the process body. Since such processes can be easily translated into efficient event-oriented models without any process-orientation overheads, one can reasonably expect to simulate such *quasi* processes faster than pure processes. The performance results shown in figure 4 demonstrate that indeed our process-oriented view implementation achieves almost the same speed as the equivalent event-oriented translation of these models. On the multi-processor runs, the rollback behavior of the PNNI models remained the same, thus making the results directly comparable. On 5 and 6 processors, the PHOLD model experienced 3% and 4% fewer rollbacks respectively on the process-oriented translation as compared to the event-oriented translation, making the process-oriented translation in fact marginally better than the event-oriented translation. This rollback behavior has been observed to be invariant across repeated simulation runs.



(a) PHOLD



(b) PNNI

Figure 4: Performance of our Implementation on Process-oriented Models as Compared to Equivalent Event-oriented Models

Large Scale Process Orientation

Using the stack reconstruction implementation, we are able to perform optimistic parallel simulations of models containing processes in excess of one million. We used a simple model, written in the TeD language, of a wireless Personal Communication Services (PCS) network, in which each sector in the PCS network is modeled as an entity, while each mobile inside a sector is modeled as a process that accesses the state of the sector entity. We used sample network configurations of a square grid containing 225 sectors along each side of the grid, giving a total of 50,625 sectors. Each sector contains 20 mobile processes, giving a total of more than a million processes. The mobile behavior is expressed as a random walk over the sector grid, superimposed by a call initiation sequence modeled using inter-call generation times and call holding times. This demonstrates that large-scale optimistic parallel simulation of process-oriented models is indeed feasible to implement efficiently.

6 CONCLUSIONS

Based on the results of our stack reconstruction approach, we observe that process-oriented views can be implemented in optimistic parallel simulations as efficiently as in conservative parallel simulations. This is possible by carefully reducing the state-saving operations to the absolute minimum. The stack reconstruction approach allows us to capture the modifications to the process-state in order to transparently support incremental state-saving. It also results in very efficient process context switches for processes with relatively small *active* stack depths. In addition, it brings with it the added benefits of reduced memory overheads, portability, and process-migratability across heterogeneous platforms. This approach can be easily implemented using a preprocessor or can be incorporated into existing simulation languages.

In models containing processes with large active stack depths, we intend to explore the tradeoff points at which other alternative implementations perform better than the stack reconstruction approach.

ACKNOWLEDGMENTS

The authors thank Christopher Carothers for helpful discussions. This work is partially supported by DARPA Contract N66001-96-C-8530 and by NSF Grant NCR-9527163.

REFERENCES

Appel A. W. 1992. Compiling with Continuations. Cambridge University Press.

Bagrodia R. L., Liao W. 1994. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. IEEE Transactions on Software Engineering, Vol. 20(4).

Booth C. J. M., Bruce D. I. 1996. Stack-free Process-oriented Simulation. In *Proceedings of 11th Workshop on Parallel & Distributed Simulation*.

Frigo M., Leiserson C. E., Randall K. H. 1998. The Implementation of the Cilk-5 Multithreaded Language. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), June 17-19, Montreal, Canada.

Koopman P. J. Stack Computers: The New Wave. On-line book at `http://www.cs.cmu.edu/~koopman/stack_computers/`.

Mascarenhas E., Rego V. 1996. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software — Practice and Experience*, Vol. 26(3).

Mitrani I. 1982. *Simulation Techniques for Discrete Event Systems*. Cambridge University Press.

Nicol D. M., editor, Special Issue on the TeD. 1998. *SIGMETRICS Performance Evaluation Review*, Vol 25, No 4.

Perumalla K. S., Andrews M., Bhatt S. 1997. A Virtual PNNI Network Testbed. In *Proceedings of the 1997 Winter Simulation Conference*, ed. C. Andradottir, K. Healy, D. H. Withers, B. L. Nelson, 1057–1064.

Perumalla K. S., Fujimoto R. M., Ogielski A. T. 1998. The TeD Language Manual. Available on-line via `http://www.cc.gatech.edu/computing/pads/ted.html`.

Poplawski A., Nicol D. M. 1998. Nops — A Conservative Parallel Simulation Engine for TeD. In *Proceedings of the 12th Workshop on Parallel & Distributed Simulation*.

Ramkumar B., Strumpen V. 1997. Portable Checkpointing for Heterogeneous Architectures. In *27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, 58–67.

Ronngren R., et al. 1996. Transparent Incremental State Saving in Time Warp. In *Proceedings of 10th Workshop on Parallel & Distributed Simulation*.

Tanenbaum A. 1995. *Distributed Operating Systems*. Chapter 4, Prentice Hall.

Whitehurst R. A., Brutocao J. 1998. Parallel Execution of Process-based Simulation Models. In *Proceedings of SCS Multiconference*.

AUTHOR BIOGRAPHIES

KALYAN S. PERUMALLA is a Research Scientist at the College of Computing at Georgia Institute of Technology, working towards the Ph.D. degree in the area of parallel simulation techniques for large-scale telecommunication networks.

RICHARD M. FUJIMOTO is a professor at the College of Computing at Georgia Institute of Technology. He has been an active researcher in the parallel and distributed simulation community since 1985.