Performance Prediction of a Parallel Simulator*

Jason Liu David Nicol Brian Premore Anna Poplawski
Department of Computer Science
Dartmouth College
Hanover, NH 03755
{jasonliu, nicol, beej, annap}@cs.dartmouth.edu

April 26, 1999

1 Introduction

There are at least three major obstacles thwarting widespread adoption of parallel discrete-event simulation (a) lack of need, (b) lack of tools, (c) lack of predictability in behavior and performance. The plain truth is that most simulation studies can be adequately done on ordinary serial computers. Parallel simulation tools are products of research efforts, and simply don't stand up to the demands of modern software engineering. The results of 20 years of research in parallel simulation reveal it to be a highly complex endevour, with performance results very much dependent on implementation details and model characteristics.

The Scalable Simulation Framework (SSF) [2] is an effort to address some these concerns. It addresses lack of need in two ways; it provides a modeling API that is attractive both for serial and parallel simulation, with parallel execution requiring no change to the model, and it targets large-scale telecommunication system modeling, an application area that requires the computational capabilities of parallelism. SSF is addressing the concern about tool quality by making software engineering a first concern, not an afterthought. It is a well-thought-out framework specification, not an implementation (although three such exist). Much more is said about this important facet in [2]. The present paper addresses the concern over unpredictable behavior. We show how we measured the internal overheads of the Dartmouth implementation of the SSF API (DaSSF), and how those measurements can be used to predict the performance of a given model, using given features of the simulator, *without having to run, or even build, the model*. The last point is crucial. Prior to our work, the best that could be done was to predict parallel performance from observed serial traces, which of course presumes that the model is built and run. Furthermore, it is an approach that does not scale to very large scale models, which require very large scale processing. We assert that the ability to accurate estimate performance *without* running a model is crucial to support rational software design, and can greatly boost one's confidence that the effort one puts into model design will yield the desired performance gains.

The significance of our work is conditioned in no small part on whether DaSSF is competitive with ordinary serial simulators. This issue is addressed in part in [2]. On large realistic network models running on a commodity Sun SMP DaSSF delivered on the order of 100,000 network events per second, per processor (a network event is generating a packet, routing a packet, or receiving a packet). As such DaSSF is as much as an order of magnitude faster than commericial network simulators. DaSSF is an outgrowth of Nops [14], which was shown to signficantly outperform TeD, GTW, Maisie, and CSIM.

What we report on here looks deceptively easy. It is not. The Heisenberg principle of performance monitoring (you cannot observe a program without affecting its behavior) prohibits very fine-grained instrumentation. Then one is left with the problem of isolating the cost of individual components of simulator operation (e.g., transfering a message from one portion of the model to another). Space and interest constraints prohibit a full description of all the methodology used to estimate simulator overheads. We hope at some point to be able to automate much or all of this process; at present, to move to a different architecture is to require that the whole study be done once again by hand. This paper is properly viewed demonstrating proof of concept. Nevertheless, the remarkable thing is that the method works. Comparisons of predicted performance with observed performance over a wide range of operating conditions usually yield estimates that enjoy 10% accuracy. Such accuracy is more than ample for the purposes of model design.

2 DaSSF

DaSSF borrows ideas from many different simulators. It resembles tools such as CSIM[15], simC[16], C++SIM[9], Awesime[7], Simpack[5], and SimKit[6] in that it is ba-

*This work was supported in part by NSF grants CCR-9625894, ANI-9808964, and DARPA contract N66001-96-C-8530.

sically a set of class libraries and a runtime system; one builds a model by writing a C++ program in a particular style. Like a number of other parallel simulators (sim++[4], Maisie[1],U.P.S.[11],TeD[12],Nops[14],Apostle[17]) it supports a process-oriented world view, although one can also (and concurrently) employ a discrete-event world view. The structural model is the common one of inter-connected entities that exchange messages. Process threads are normally associated with entities. Like TeD, Nops, and Maisie, DaSSF implements its threads by instrumenting user code at the source level, and like these tools it uses a source-to-source translator. DaSSF approaches synchronization among parallel processors conservatively, using global time windows. In this it resembles U.P.S. and Nops.

We've adopted C++ for the purpose of achieving high performance while still enjoying the benefits of object orientation. A significant part of our overall project is to identify common model patterns in networking, and develop reusable model libraries based on those patterns. Our early efforts are reflected later in this paper when we report on the performance of a routing benchmark.

In the parallel simulation realm, DaSSF most closely resembles TeD, U.P.S., and Nops. DaSSF differs from TeD by minimizing the amount of extra required structure and macros needed to express a model. For instance, TeD puts into its language structure essentially the same relationship as an abstract base class has in C++ with a concrete class derived from it. The DaSSF philosophy is to let C++ do what it does well, and to minimize the syntactical additions. DaSSF differs from Nops both in being built entirely in C++ (Nops uses C and the Cilk threads system which is also in C), and in providing its own API (Nops is a target for the TeD API). A very significant difference between DaSSF and U.P.S. is that the latter is built on top of a commercial simulation package whose internals are not accessible. Some of DaSSF features can be emulated in CSIM, but at significant cost. Other features could not be supported at all e.g., automated memory and workload balancing.

In the network simulation world DaSSF stands along side SMURPH[3] and SimKit[6] as simulators targeting networking, using a C++ base class approach. The essential differences between DaSSF and SimKit (both of which run in parallel, unlike SMURPH) are that SimKit is specifically event-oriented. DaSSF has a richer API, and offers a somewhat more general framework within which the modeler can specify and control model concurrency. Of course, it may well be that the price DaSSF pays for its additional flexibility is diminished raw performance relative to SimKit.

DaSSF's overriding design goal is to support the efficient simulation of very large networks. This goal governs the way DaSSF synchronizes and it governs the way it manages memory. It motivates DaSSF's features, and it motivates the present study. Only by understanding the location and magnitude of these overheads can we confidently build models that suffer the overheads when it is expedient to gain the associated features, and avoid them when it is not.

3 Basic Concepts

The DaSSF class library consists principally of **Entity**, **Process**, **Event**, **inChannel**, and **outChannel** classes. Normally entities are containers for state data, and for processes that manipulate that data. Channels connect entities. Code associated with an entity (e.g. a process) may send an event—a message—to another by "writing" it to an outchannel. Likewise, code associated with an entity is triggered to execute by receipt of an event on an inchannel.

Naturally, all of the constructs named above are base classes. The critical **outChannel** methods are `write`—used to send an event out—and `mapto`, that associates a outchannel-inchannel pair as endpoints of a communication channel. An outchannel may be multicast (i.e., mapped to multiple inchannels), and an inchannel may be the target of multiple outchannels. A simulation delay is inserted between the time at which the `write` is executed (the "send-time"), and when the event appears at inchannels mapped-to the source outchannel (the "receive-time"). That delay is comprised of a *minimal channel delay* which is declared as part of the outchannel constructor call, and an additional delay specified as part of the `write`. Either of these delays may be zero. Each inchannel and outchannel is declared to be "owned" by an entity whose identity is passed in the channel constructor.

The critical methods for the Process class are the "wait" functions. `waitFor` suspends the process for the period of time given as an argument. `waitOn` takes one or more inchannels as arguments; it suspends the process until the next simulation time that an event appears on any one of the named inchannels. `waitOnFor` is a combination of these two; it is a `waitOn` that times-out after a specified interval of simulation time. A process is started and runs by calling the constructor of a class derived from Process.

DaSSF is flexible concerning exposure of concurrency. For example, in TeD the **Entity** is the unit of concurrency. Synchronization and scheduling are conducted largely as though different entities execute on different processors. The same is true of Maisie, and its successor PARSEC. However, sim++ allowed one to group logical processes into "clusters", and DaSSF does as well using its notion of *alignment*. Methods exist to align one entity with another. This is an explicit statement by the modeler that all simulation activity associated with the aligned entities will be performed in monotone non-decreasing time-stamp order. In DaSSF parlance they exist on a common "timeline". Every entity resides in some timeline, different mechanisms exist for establishing alignment. The `makeIndependent` method causes the associated entity to initialize a new timeline, with only itself as a member. `Entity`

method `alignTo(Entity *tgt)` puts the calling entity on the same timeline as the argument. If an entity is not explicitly aligned, it defines its own timeline.

If two non-aligned entities can communicate through a channel, then that channel must have a non-zero minimum delay. This requirement gives DaSSF the lookahead it needs for its form of conservative synchronization.

Control over alignment gives two advantages. Each timeline has its own event-list. For a given model, increasing the number of timelines increases the number of event-lists representing the aggregate simulation load, reduces the average size of each eventlist, and so lowers the per-event access cost. On the other hand, increasing the number of timelines increases the synchronization-window overhead of initializing timelines, and can reduce lookahead by exposing inter-timeline channels with lower minimal delays. The tension between these facets can be explored to optimize performance, by making minimal changes to the code to affect alignment. A second advantage is that co-aligned entities can safely interact with each other through means other than channels, e.g., direct method calls. An entity can test whether another is co-aligned before interacting with it in such a fashion. This sort of flexibility simplifies model expression by expanding the repertoire of language constructs.

At present, alignment and containment relationships are statically declared during model initialization. We will soon be investigating dynamic and automated alignment.

4 Implementation of Features

4.1 Threading

Implementation of threads is the trickiest part of any process-oriented simulator. For this reason some simulators (e.g., C++SIM) avoid the problem altogether by employing an existing threads package. While there are several obvious advantages to this approach, high performance and efficient use of memory isn't one of them.

DaSSF follows other simulators (e.g., TeD, Nops, Maisie) in implementing thread behavior by instrumenting source code using a source-to-source translator. The mechanism DaSSF uses to implement threading resembles that reported for TeD in [13]. For the reader to appreciate the cost of a DaSSF context switch we need to describe the structure of a transformed code. Only certain methods need to be transformed, precisely those which may be on the call-stack when a thread suspends. We call such methods *procedures*. A procedure is transformed at its entry point, at references to its state variables, at subroutine calls it makes to other procedures, and at return statements. A DaSSF defined *p-frame* represents a procedure, containing re-entry information, state variables, and a pointer for return values. The p-frame just reflects what the procedure's ordinary run-time stack frame would contain, but the p-frame is in the run-time heap, not the run-time stack. When a thread suspends, a stack of p-frames contains its state, just as would the run-time stack at that same instant.

Within the body of the procedure, references to local state variables are transformed to refer to their p-frame equivalents. "return" statements are transformed to call a DaSSF function that handles procedure returns.

A procedure's p-frame is created *by the procedure that calls it*, executing instrumentation code inserted automatically for this purpose. A p-frame derives from a base class **Procedure**; there is a unique derived class for every procedure. The p-frame is created by calling its class constructor, passing the procedure input arguments to the constructor for storage. This approach is essential when the called procedure is a virtual method, for then the called procedure constructor will also be virtual and will have been made (by the translator) a virtual member function of the procedure's class.

After each procedure call the DaSSF translator inserts code (into the caller's code body) that immediately executes a return if the current thread is suspended; this is followed by a unique label to identify the return point. A code for this label is stored before the call is made. The entire body of the procedure is wrapped in a "switch" statement; there is a "case" statement for each entry point code, the action associated with a code is simply a "goto" the associated label.

To see how all of these works together, imagine that a thread executes a **wait** command that suspends it. The thread data structure points to the stack of p-frames that reflect its state. The thread is marked as suspended, and the next thread to run on this timeline is identified through a call to the timeline's *microscheduler* (about which we'll say more later). If such a thread exists it is marked as the next thread to run, otherwise a thread from a different timeline is so marked. It remains to clean up the run-time stack and to dispatch the next thread. A sequence of returns is initiated, using return addresses on the run-time stack, bringing execution to statements that simply return if the thread is suspended. This sequence stops when it reaches the code body of the thread dispatcher, which dispatches the thread already identified as being next to run.

Imagine now that the newly dispatched thread had been suspended, and is now being re-animated. The procedure at the top of the thread's stack of p-frames is called, the entry point switch statement directs it to execute at code following the call (a **wait** statement) that suspended it. Supposing that the procedure now executes through to a transformed return statement, it calls a DaSSF routine to handle the return. That routine pops the top element of the thread's p-frame stack and reclaims that memory, and then just returns—back to the thread dispatcher, which dispatches the next thread to run—the same one—but this time a different procedure is

entered to simulate the normal return a normal subroutine call would provide.

4.2 Dynamic Channels

When a process blocks on a **wait** statement it specifies a set of channels it to which it becomes sensitive. The next arrival on any of those channels will unblock the process, at the simulation time of the arrival. The DaSSF API specifies that a process may block on a "static" set of channels, or a dynamic set of channels. In the former case, after an arrival the process remains sensitive to arrivals on all channels in the static set, in the latter case it loses its sensitivity to all channels in the dynamic set. There are overhead costs of setting up and tearing down relationships between processes and dynamic channels, costs that should be understood if one contemplates using dynamic channels. Another consideration is that multiple processes may be concurrently blocked on a channel; when an event shows up, all the processes blocked on it must be executed.

Tear-down costs are incurred in the timeline microscheduler. A timeline's event list contains arrivals of events on inchannels, scheduled function invocations, and time-out events for processes suspended on `waitFor` statements. The cost of executing the microscheduler depends on the type of event. For a time-out or function evaluation there is little to do. Some complexity arises when the event represents a channel arrival, for now we must find the next process to run and must manipulate some data structures.

The process being activated may have put itself on the waiting lists of other channels. Under DaSSF semantics, when a process unblocks from a **wait** statement it must be removed from the waiting lists of all the channels specified as arguments to that statement. Since each of these channels may have a list of waiting processes, the correct element of each such list must be removed. DaSSF is clever enough to avoid searching, but there are costs of updating cross-linked data structures. One last cost occurs if the process blocked on a **waitOnFor** statement; there is then a time-out event which must be canceled.

The channel constructs aid significantly in one's ability to construct modular models, and realize code reuse. Channels can be viewed as providing a level of indirection for communication; an entity's functions need not know the source of an event, nor need they know the destination of events they write. Entity code bodies can concentrate on local behavior; the model's global behavior is the aggregation of many such local behaviors.

4.3 Simple Processes

It is well-known that process-oriented models run more slowly than functionally equivalent discrete-event models. As we are mindful of the needs of some simulations to achieve the very highest performance, SSF is engineered to support a discrete-event paradigm along side of the process-oriented paradigm. SSF includes the notion of a *Simple Process*, where the only place a **wait** statement occurs is the last statement executed in the code body. This feature allows the implementation to dodge process suspension costs, as well as other optimizations. From the modeler's point-of-view a simple Process is a piece of code that simply reacts to the receipt of an Event, or a scheduled time-out. In this sense it is simply an event-handler.

5 Experiments

Next we describe a set of experiments designed to reveal DaSSF's underlying implementation costs, on an SGI Origin2000 shared-memory multiprocessor. Our Origin2000 has 180MHz R10000 CPUs, 1Mb of L2 cache for each CPU, and 4Gb of main memory. We use the Origin2000's hardware counters to measure code execution durations.

We first explore the cost of context-switching, considering three different ways it can occur. Next we measure the cost of creating and deleting the two most dynamically used DaSSF classes, **Event** and **Process**. We assess the cost of calling an instrumented procedure, and evaluate the performance differential between using static and dynamic channels. We examine the performance differential between process-oriented and discrete-event oriented execution, measure how event-list costs vary with increasing size, and measure costs incurred at synchronization windows.

5.1 Context Switching

The cost of a context switch depends considerably on the mechanism invoked to identify the next thread to run; it can also depend on the number of processes on a timeline at the point of the switch. Our experiments quantify these differences. Except when noted, all experimental activity occurs on one timeline, and hence there is no window synchronization activity interfering with the measurements.

One way a context switch occurs is if a process "times-out" on a `waitFor` call. We measure this cost as a function of N processes waiting on time-outs, by first creating N processes, all on one timeline. Each process enters a loop where it randomly samples an exponential e (with mean 1.0), and does a **waitFor(e)**. This staggers the times at which processes awaken, and keeps N time-out events on the timeline's event list. The loop is executed many many times; the process is no longer run once the simulation time advances past the simulation's termination time. Wallclock time is measured immediately before process 0 enters the loop, and is measured immediately after it exits the loop. The difference is divided by the total number of "wake-ups" that occured. However, this cost includes that of sampling an exponential. We measure that cost in a similar fashion, and subtract it from the average cost computed in the first experiment.

The second set of experiments assesses the context switch cost related to blocking on a channel. Entirely dif-

ferent paths through the timeline microscheduler are taken than with a time-out, so different costs are incurred. To assess this cost we constructed N processes, N outchannels (each with delay 1), and N inchannels. Each process i declares itself to be statically sensitive to inchannel $(i-1) \bmod N$, and (with the exception of process 0) does a `waitOn` on that channel. When an event arrives on that channel, process i immediately writes out on channel i, which has been mapped to inchannel $(i+1) \bmod N$. The experiment is triggered after initialization with process 0 writing an event out on outchannel 1. A large number of iterations around the loop are timed, and the average cost per context switch is computed. The experiment included the cost of creating and deleting one **Event** per context switch and the cost of writing an event through an outchannel. These costs are subtracted from the experimental per-context average to arrive at the final context switch cost. Bear in mind that these costs involve static channels; subsequent experiments will give us correction factors for dynamic channels.

The third set of experiments measures the context switch cost for processes that are all waiting on the same channel. The experimental framework consists of one *producer* process, one outchannel, one inchannel, and N *consumer* processes. The outchannel has delay 0, and maps to the inchannel. The body of the producer process loops, with each iteration it waits for 1 unit of simulation time, then generates and writes an event to the outchannel. The consumer processes all initially declare themselves to be statically sensitive to the one inchannel, and then enter a loop which does nothing but call **waitOn**. When an event arrives, the first process on the channel's wait list is activated. It simply calls **waitOn** again, suspending it, and control passes again to the timeline's microscheduler which observes that there remain processes sensitive to the last event evaluated, finds the next as-yet-executed process sensitive to the channel, and activates it. This sequence repeats through all consumer processes. No simulation time elapses between different processes' activations, so after all consumer processes have run once, simulation time advances one unit due to the producer's **waitFor** call, and the whole sequence repeats itself. To obtain the context switch cost we time a large number of iterations. From this cost we subtract the contribution of the producer and the first consumer to awaken, measured by the running time of an experiment with one producer and one consumer. The difference is the context switching time of the remaining processes. We obtain the average context switching cost by dividing this difference by the total number of context switches associated with those processes.

These three forms of context switching each depend on parameter N. Figure 1 plots the average context switch cost in micro-seconds for each form, as a function of $N = 2^k$, for $k = 1, \ldots, 16$. The most immediate impression is that

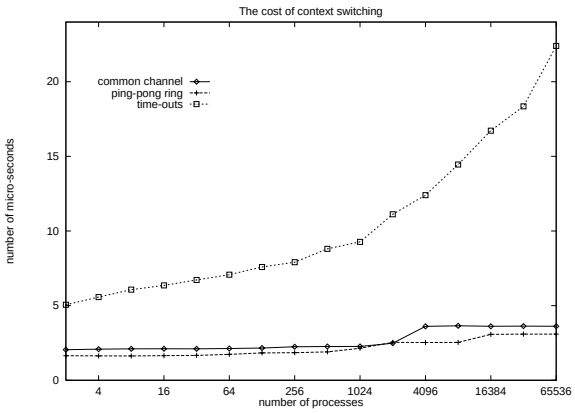

Figure 1: Context-switch times in micro-seconds, for three types of context switches

the cost curve for the **waitFor** experiments dominates the others. The reason for the difference is that in the other two sets of experiments, the event-list has at most one element at any instant; the comparison mechanisms of the event list are not exercised. For the **waitFor** experiments however, there is nothing but event-list manipulation and suspension infrastructure. Its cost rises slowly as the size of the event-list (a splay-tree) increases. We speculate that the rising behavior of the ping-pong benchmark is related to memory management, evidently with increasing N additional instructions are being executed somewhere, and memory routines are the best bet.

The essential information to take away from this graph is the overall scale of the context switching costs and that event-list costs contribute significantly. We will use this data when estimating application performance.

5.2 Dynamic Object Costs

Both events and processes can be dynamically created and destroyed. Since this exacts a cost, it is important to know what that cost is.

To measure the overhead of creating and destroying an event we time a loop whose body creates a large number of events, and then destroys them (in a different order than they were created). So measured, the average cost was measured to be 242 instructions (or 1.94 μ-seconds);

To measure the cost of creating and destroying a process we augment the `waitFor` context switch experiment so that a process awakens to create a new `Process`. The process body is empty however, so no sooner is the process created than it is destroyed as well. The difference in running time between this experiment and the `waitFor` context switching experiment reveals the sought cost, measured as 1313 instructions (or 9.5 μ-seconds).

5.3 Procedure call overhead

Calling an instrumented procedure exacts a cost going in, and coming out. Going in the cost includes creating a p-frame and following an entry code to an entry point. The cost coming out includes a test for thread suspension; if the thread is suspended there will later be an additional cost of re-entering the called procedure. In both cases there is a cost of reclaiming the p-frame space.

To measure these costs we added a stack of procedure calls to the `waitFor` context switching measurement code (using just two processes). Immediately preceding the `waitFor`, a ten-deep stack of procedure calls (to procedures that do nothing but call other procedures) is executed. To emulate the low cost returns, the thread is not suspended, so the stack of p-frames unravels immediately. For a given experiment we can subtract the running time of the original `waitFor` code from that of the augmented code, to get Procedure overhead. Dividing through by the number of times procedures are called identifies an overhead of 520 instructions (3.5 μ-seconds), The same technique can be applied to obtain the *additional* cost per procedure when the thread is suspended. For this we move the **waitOn** to be at the bottom of the procedure call chain. The additional cost is 106 instructions (0.7 μ-seconds).

5.4 Dynamic Channel Overhead

When a process uses dynamic sensitivity to channels it incurs a cost of establishing the sensitivity when `waitOn` is called, and de-establishing it when an event arrives on one of the named channels. Both costs are linear in the number of channels named. To measure this cost we modified the producer-consumer context-switching experiment, so that each of the customers declares sensitivity to M inchannels. When we run this scenario with static sensitivity, we have exactly the behavior observed in the context-switching experiment. The additional runtime observed by executing it with dynamic sensitivity reveals the extra costs involved. That difference, suitably normalized, gives the extra cost, per inchannel, of using dynamic sensitivity instead of static. The cost reflects the overhead of one setup and one teardown on one channel. It is measured as 336 instructions (2.25 μ-seconds). These figures are relatively constant across three orders of magnitude of N.

5.5 Cost of Process Orientation

DaSSF's ability to use both discrete-event and process-oriented paradigms provides a fair framework in which to compare the relative costs of these two approaches. In doing so we need to remember that the DaSSF discrete-event approach still enjoys—and pays the price for—the modularity afforded by its modeling architecture, and the generality afforded by its base-class approach. We cannot expect a DaSSF discrete-event model to achieve the performance of a simulator hand-crafted to a single application.

In the DaSSF framework the performance difference between paradigms is due to threading overhead, and complexities of dealing with dynamic channels in the microscheduler. This difference is bound to be application specific; if the essential workload granularity is large then differences in how one identifies that workload will contribute less to the overall performance. Recognizing this, we've written discrete-event versions of our context-switching benchmarks. On the one hand, the process-oriented versions do not incur the substantial costs of procedure calls, on the other hand, their computational grain is extremely small.

The **waitFor** context switching experiments uniformly (for all N) ran 34% faster using simple Processes; the ping-pong ring experiments ran 60% faster. The producer-consumer experiments varied with N, runs with small N were 40% faster, increasing steadily to about 100% faster as N grows. The sensitivity to N is a result of amortizing the fixed cost of generating an event and sending it down a channel over increasingly more Processes.

5.6 Event List

Each timeline has its own event-list, implemented as a splay-tree. As event-list costs contribute substantively to the overall execution, we ran an standard experiment to estimate the cost of inserting and deleting an event.

Like many before us we implemented the hold-model, where the event list is initially seeded with N events, and then a loop is executed where each iteration the minimum-time event is removed, a random value is added to its time-stamp, and it is re-inserted. We time the loop executing many many ($O(10^5)$) times, and take an average to obtain a cost of inserting and removing one event. We remove from this cost the contribution of calling the random number generator, obtained in a similar fashion.

The table below gives the average cost in micro-seconds of each insertion and deletion pair, over 6 orders of magnitude of event list sizes.

Size of Splay Tree						
10^0	10^1	10^2	10^3	10^4	10^5	10^6
0.42	1.2	1.9	2.72	4.08	11.86	19.4

Insert/Delete Cost, in μ-seconds

This data clearly shows the effects of cacheing. While the number of issued instructions grows as one expects for a splay-tree, the execution times for lists of size 10^5 and 10^6 are "too big" relative to earlier sizes. In an application we may see the on-set of cache effects for even smaller event-list sizes, because the cache will contain non-event-list data as well.

5.7 Cost of Synchronization

DaSSF synchronizes just as did Nops [14]. All processors synchronize every c units of simulation time, where c is the smallest minimal delay on any outchannel mapped to

an inchannel on a different timeline. At the end of such an epoch the processors engage in a barrier synchronization. Following this they each move all events for the next epoch from the processor's event-list (events generated for future epochs all go here) into exchange buffers, and again do a barrier synchronization. Following this, each processor moves events from the exchange buffers into timeline event-lists, and then calls the microscheduler for each timeline so loaded.

The first step—barrier synchronization—has a cost that depends on the number of processors but is independent of the size of the model. The table below gives the cost of the logic leading to and including a barrier synchronization, in micro-seconds, as a function of the number of processors used. The barrier mechanism used is not especially optimal, but considered at the resolution of other things in the simulation, 100 μ-seconds isn't much.

1	2	3	4	5	6	7
4.9	20.9	35.0	54.7	71.6	100.6	146.8

Cost of barrier synchronization, in μ-seconds

The second aspect of synchronization is the movement of events between processors at a synchronization window boundary. Measurements show this cost to be approximately 1 μ-second per event, per window.

The last step moves events to timeline event-lists and initializes each timeline scheduler. We actually need not concern ourselves with this. The timeline initialization is small relative to everything else. The cost of moving events into event-lists is substantial, but we can account for its cost elsewhere. For instance, when we model the cost of sending an event, we'll include the associated cost of putting it on the event-list, as well as taking it off. For events that cross synchronization window boundaries, the insertion happens at the window initialization, not at the point the write is executed.

6 Application

We now apply some of the measures we've obtained to estimate the performance of a reasonably complex network simulation, running on 5 processors. We find that in this case a good back-of-the-envelope performance estimate can be constructed by adding up various costs. We also experimentally confirm earlier discussions about the effect of alignment decisions.

We consider a network model that organizes 5670 "hosts" into 150 LANs (local area networks); there are 1000 routers. Each host emits bursty traffic, targeting randomly selected destinations. The routing is handled using routing tables established with two protocols, BGP [8] between LANs, and OSPF [10] within a LAN; the simulation reads these routing tables and the associated topology at startup. The simulation uses IP addressing, with IP address vectors being mapping into unique integers. A routing decision (given a destination, which output port should receive the packet) is made using a radix search tree lookup (a standard technique in routing).

The computational workload consists of packet generation, and routing. We isolated critical code blocks and measured the average cost to be 2.5 μ-seconds to generate a packet and its destination, and 2.0 μ-seconds to route a packet. Next we couple these numbers with DaSSF overheads to estimate overall performance.

The total cost of generating a packet includes the workload (2.5 μ-seconds), plus a context switch and a channel write. The context switch is of the **waitFor** variety, and so we'll use the data in Figure 1 to quantify context switch cost $C(S)$, S denoting the size of the event-list. A channel write costs 1.2 μ-second plus an event-list insertion and deletion. We denote the event-list cost by $E(S)$, and will quantify it using the data from §5.6. The cost of generating a packet is thus $3.7 + C(S) + E(S)$.

The total cost of routing a packet includes a channel-based context switch which we take to be 2.5 μ-seconds (from Figure 1). It includes a routing lookup of 2.0 μ-seconds. It includes a channel write, modeled as $1.2+E(S)$. It also includes a 1 μ-second cost of transferring the packet between processors at exactly one synchronization window boundary. In the present scenario, every packet is so transferred exactly once per router, for a per-router cost of $6.7 + E(S)$ μ-seconds.

The routing topology is hierarchical. At the lowest level the number of hosts per LAN and the interconnections vary somewhat. We estimate it takes takes 3 hops on average to exit the LAN. The higher level topology is more regular, it has a diameter of 5 and with the randomized destination selection a packet should travel a mean distance of 2.5 hops before reaching its LAN. Another 3 hops within the LAN delivers the event. Thus we estimate that a packet will pass through 8.5 routers on average (this estimate was then verified by measurement). Summing it all up we arrive at a per-packet execution cost estimate of $3.7 + C(S) + E(S) + 8.5(6.7 + E(S)) = 60.65 + C(S) + 9.5E(S)$ μ-seconds.

There are 5670 hosts, generating packets at an average rate of 5000 packets/simulated-second. The latency between connected routers is 50 msec. Assuming the workload is distributed uniformly over each of 5 processors, we obtain an estimate on the ratio of real execution time required to advance one unit of simulation time (in parallel) by taking the product of (i) the number of hosts per processor (in this case, 1134), (ii) the number of packets generated per simulated second, per host (in this case 5000), and (iii) the execution time per packet (in this case, $60.65 + C(S) + 9.5E(S)$ μ-sec).

We've delayed quantifying the event-list cost because the size S depends on the number of timelines. Natural groupings within the model suggest alignments into 5,

55, 135, 510, and 1005 total timelines. Assuming events are distributed evenly among timelines and assuming that a packet is represented by one event, the total number of events in event-lists during a synchronization window is 1417500 events. Bearing in mind that the event-list size is not steady, it ramps up at the beginning of the window and drains away, we'll take half this number as the average size. For the various timelines considered, this gives values of $S = 141750, 12886, 5250, 1389, 705$.

The table below compares the measured cost per unit simulation time (in seconds), with those predicted by taking $C(S)$ from Figure 1, approximating $E(S)$ from the splay-tree cost table (interpolating roughly from the table when needed), and adding in the event transfer costs.

	Number of Timelines per Processor				
	1	11	27	102	201
Measured	793	631	588	543	522
Predicted	1121	654	579	513	486

Real seconds per unit simulation time

Considering the coarseness of the estimates, the agreement is remarkable. The key thing our estimates ignore is the additional cost of synchronization delay at a barrier due to load-imbalance, this cost is a function of the way the model is partitioned. Most of the discrepency can be explained by synchronization delays we don't model. At the data point where there is one timeline per processor the measured behavior takes a steep jump. This results from the same jump in event-list costs, when a critical threshold of memory useage pushes behavior into another regime of cacheing behavior. While the model did not capture the effect exactly, it is clear from the measurement data that indeed a jump in costs occurs at this point.

To push the envelope a bit we estimated the performance of a different configuration of this code. We re-wrote it to use simple processes, and to eliminate an unnecessary router layer within a LAN. Now we estimate the cost of generating a packet as a 2.5 μ-second generation cost, plus the 1.2 + $E(S)$ write cost, plus the cost of scheduling and executing another packet generation event. The mechanics of that are virtually identical to a channel write, so we'll use the same cost estimate to arrive at a packet generation cost of $4.9 + 2E(S)$ μ-seconds. The cost of routing a packet is the 2 μ-second routing cost, plus a $1.2+E(S)$ channel write cost, plus a 1 μ-second window transfer cost, plus a function evaluation, measured earlier at 0.33 μ-second, for a total router cost of $4.53+E(S)$. Having cut out two router layers, the total work associated with a packet is $34.34 + 6.5E(S)$ μ-seconds. The table below compares measured and predicted costs of advancing simulation time.

	Number of Timelines per Processor				
	1	11	27	102	201
Measured	726	354	356	308	293
Predicted	763	391	339	296	276

Real seconds per unit simulation time

Once again we see that our estimates track measurement well. This exercise clearly validates the utility of knowing implementation costs.

Although the point of this data has been to demonstrate application of the implementation costs, there are other interesting points about the data. The first is that the structure of DaSSF exploits model separation to reduce event-list costs in a way that an ordinary simulator with one event-list would not. The performance implications are substantial. In the table immediately above there is over a factor of two difference in performance between the one timeline-per-processor case and the case with 201 timelines. DaSSF has a "serial" mode where it acts as would an ordinary serial simulator by ensuring that there is but one synchronization window and that all events go through a common event-list. In our experience the performance of this mode is always worse than "parallel" DaSSF on one processor, with windows and multiple timelines. Clearly an ability to easily adjust exposure of concurrency can yield considerable performance benefits. This point has been made before by others and for different reasons, but bears repeating. A second point is that the second model runs about 80% faster than the first; much of that contribution is due to use of the simple Process mechanism. And the final point to report is that DaSSF is delivering excellent parallel performance on these runs, better than 90% processor utilization (the tables don't imply this, we just report it).

7 Discussion

At first glance the back-of-the-envelope method for performance prediction would seem to require intimate knowledge of the application. A basic knowledge of how the simulator works is surely required, but pieceing together new performance models is no more complex than the one just outlined. Once the knowledge of simulator behavior is obtained, doing what we have done for a different model is straightforward. Another point is that complex behavior of user model code (e.g., routing table look-up) is abstracted down to a single execution cost. The code we modeled here has thousands of lines of code; we represented that code with a small handful of execution costs. Knowledge of how the submodel elements will interact with each other, at some level, is required, but that is a requirement for *any* predictive performance approach.

It will also be noted that our modeling approach assumed perfectly balanced workload. This is not an unreasonable assumption for DaSSF, as it provides transparent automated

dynamic load-balancing, with negligible overhead costs.

8 Conclusion

The SSF project is addressing the need of the telecommunications community for high-performance, scalable simulation technology. That technology involves both high-performance simulation kernels, and strict software engineering principles. DaSSF is a tool resulting from this project.

One of the requirements for principled design of high-performance software is model-specific performance prediction, as part of the design process. To date such a capability has been missing from parallel simulation technology. We believe that the complexity of most approaches to parallel simulation prohibit such prediction. However, DaSSF's approach to synchronization is extremely simple, and for the targeted application class, very effective. This simplicity encouraged us to attempt a high-level approach to performance prediction, and this simplicity proved in the end to make that prediction accurate.

References

[1] R. Bagrodia and W.-T. Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Trans. on Software Engineering*, 20(4):225–238, 1994.

[2] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, 1(1):42–50, Jan.-Feb. 1999.

[3] W. Dobosiewicz and P. Gburzynski. SMURPH: An object-oriented simulator for communication networks and protocols. In *Proceedings of MASCOTS'93*, pages 351–353, 1993.

[4] D. Baezner et al. Sim++: The transition to distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 211–218, 1990.

[5] P. Fishwick. Simpack users guide. Technical report, University of Florida, 1995.

[6] F. Gomes, S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington. SimKit: A high performance logical process simulation class library in c++. In *Proceedings of the 1995 Winter Simulation Conference*, pages 706–713, December 1995.

[7] D. Grunwald. Awesime users guide. Technical report, University of Colorado, 1992.

[8] T. Li and Y. Rekhter. A border gateway protocol 4 (BGP-4). Technical report, Internet Engineering Task Force, March 1995.

[9] M.C. Little and D.L. McCue. Construction and use of a simulation package in c++. *C User's Journal*, 12(3), March 1994.

[10] J. Moy. OSPF version 2. Technical report, Internet Engineering Task Force, April 1998.

[11] D. Nicol and P. Heidelberger. Parallel execution for serial simulators. *ACM Trans. on Modeling and Computer Simulation*, 6(3):210–242, July 1996.

[12] K. Perumalla and R. Fujimoto. Efficient large-scale process-oriented parallel simulations. In *Proceedings of the 1998 Winter Simulation Conference*, December 1998. to appear.

[13] K. Perumalla and R. Fujimoto. Efficient large-scale process-oriented parallel simulations. In *Proceedings of the 1998 Winter Simulation Conference*, pages 459–466, December 1999.

[14] A. Poplawski and D. Nicol. Nops: A conservative simulation engine for ted. In *Proceedings of the 1998 Workshop on Parallel and Distributed Simulation*, pages 180–187, June 1998.

[15] H. Schwetman. CSIM: A C-based, process oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.

[16] S. Toh. SimC: A c function library for discrete simulation. In *Proceedings of the SCS Multiconference on Simulation in Engineering Education*, pages 18–20, January 1993.

[17] P. Wonnacott and D. Bruce. The APOSTLE simulation language: Granularity control and performance data. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 114–123, June 1996.