A (Condensed) Parametric Study of Optimistic Computation in Wide-Area, Distributed Environments

Craig A. Lee
Computer Systems Research Department
The Aerospace Corporation, P.O. Box 92957
El Segundo, CA 90009
lee@aero.org

Nikos Chrisochoides
Computer Science Department
College of William and Mary
Williamsburg, VA 23187
nikos@cs.wm.edu

Abstract

This paper explores the use of optimistic computation to improve application performance in wide-area distributed environments. We do so by defining a parametric model of optimistic computation and then running sets of parameterized experiments to show where, and to what degree, optimistic computation can produce speed-ups. The model is instantiated as an optimistic workload generator implemented as a parallel MPI code. Sets of experiments are then run using this code on an EmuLab system where the network topology, bandwidth and latency can be experimentally controlled. Hence, the results we obtain are from a real parallel code running over a real network protocol through emulated network conditions. We show that under favorable conditions, many fold speed-ups are possible, and even under moderate conditions, speed-ups can still be realized. While generally optimism provides the best speed-ups when network latency dominates the processing cycle, we have seen cases (with a 90% probability of success) when latency is only 1/6 of the processing cycle yet produces break-even relative performance and 85% of "local" performance. The ultimate goal is to apply this understanding to real-world grid applications that can use optimism to tolerate higher latencies.

1 Introduction

Optimistic, or speculative, computation is a well-known performance technique that has been applied on many different scales, in many different areas. At the hardware level, *speculative branch execution* is used to prevent bubbles from getting into an instruction pipeline. At the system level, optimism can be used to do *speculative cluster scheduling* to service applications such as speculative DNA sequencing [13]. At the computational domain level, optimistic simulation has been long known to produce performance benefits [10, 9], under the right conditions, and is still finding applications today [18]. Of course, the key issue for optimistic computation is that it can produce both performance enhancement and degradation, depending on how often the optimistic course of action is incorrect, and how severe the related performance penalty is.

The development of grids and other wide-area environments with constrained bandwidths and high latencies actually presents another arena where optimistic computation can be applied. Clearly, loosely coupled applications will be more suited to these environments, e.g., those applications and systems that essentially require resource integration but with lenient performance requirements. This does not, however, mean that we should never consider the issues faced by more tightly coupled applications on the grid. The better that latency can be tolerated, the better that a range of applications will have acceptable performance. Besides optimistic discrete event simulation, other applications that involve the exchange of control information could benefit from better latency tolerance. For example, a parallel computational fluid dynamics code that exchanges varying parametric data between neighbors may be a candidate for parameter estimation techniques, but a parallel mesh generation code that exchanges proposed new mesh points between neighbors may be a candidate for optimistic execution [5].

Hence, the research issue becomes how well can such heterogeneous latency hierarchies be tolerated? How much latency can be tolerated and under what conditions? When is a net gain in performance realized? Only after we understand these issues can we understand what level of coupling is feasible in grid applications.

The goal in this paper, therefore, is to understand quantitatively, as best as possible, how optimism can be used to improve performance by tolerating latencies in wide-area, distributed environments, e.g., grids. The approach we have taken is to define a simple model of optimistic computation

based on message-passing, and then instantiate this model in an optimistic workload generator written in MPI. We then ran this generator in different node topologies on an EmuLab system [22], where the network bandwidth and latency could be experimentally controlled. This allowed us to construct sets of parametric experiments to compare the performance of optimistic and non-optimistic computations. While there are many issues concerning the design of the model and the representation of parameters, this enabled us to make this preliminary investigation into optimistic grid computation.

We begin by reviewing some background material and related work. We then introduce the model of optimism, followed by the experimental approach and results. We conclude with a discussion.

2 Background and Related Work

Optimistic computation is just one method to tolerate latency and improve concurrency in a parallel or distributed environment. Other common techniques include overlapping communication and computation, parameter estimation, data compression, and redundant computation (e.g., ghost zones) between communicating nodes. Such techniques were nicely demonstrated for a distributed parallel solver in [1].

Bubenik and Zwaenepoel formalized the semantics of optimistic computation by modeling computations as program dependency graphs, and defining transformations that can produce an optimistic program from a pessimistic one, while preserving the semantics [3]. This work was used to derive both an optimistic method of fault tolerance based on message logging and checkpointing, and also an *optimistic make* that monitors a file system for out-of-date targets and brings them up-to-date before a *make* request is issued [4].

Optimism has also been used for parallelizing compilers on parallel machines. In [14], rather than use mutual exclusion locks to protect critical sections, optimistic synchronization primitives are used. Most parallelizing compilers approached the problem by doing extensive, static, compile-time analysis to identify critical sections. In [2], transformations are defined for object-oriented programs that enable objects to do causality violation detection and roll-back, while preserving the semantics of the sequential code.

Optimism is also widely known for its use in parallel, discrete event simulation. Time Warp [10] is generally regarded as the first optimistic simulation framework. Many other systems followed, such as SPEEDES [17] and WarpIV [16] that addressed issues such as computing global virtual time and incremental state saving to support roll-back.

The parametric studies we pursue in this paper are based on message-passing and relevant work has been done here as well. In the *active messages* concept, *handler routines* are not allowed to block or contain arbitrary user code. When using *optimistic active messages*, however, this restrictions can be relaxed [20]. Optimistic message passing has also been done using PVM in HOPE system [6, 11, 7]. There has also been a *CORBA Optimistic Programming Environment (COPE)* that improved on HOPE by hiding the optimistic machinery through object encapsulation [12]. Optimistic computation has also been done in Java with the E Extensions [8] that provide for messages that are simply “sent” with status being returned in a channel object. Additional optimistic, distributed computation has been done in Java – the *Code Undo* project approaches optimism by “undoing code” rather than “undoing messages” [21]. All in all, the value of optimistic computation in distributed environments has clearly been recognized [19].

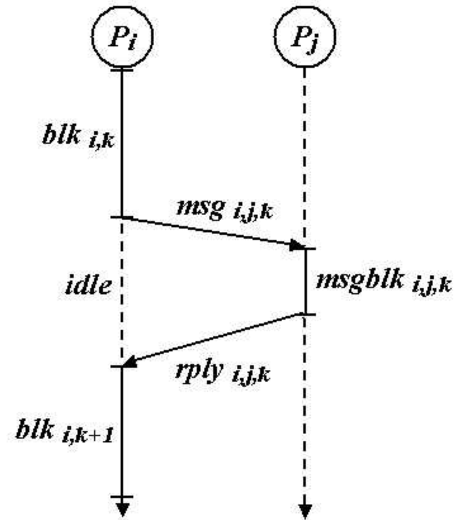


Figure 1. Workload Model.

3 A Model of Optimistic Computation

To investigate the issue of optimistic computation in distributed environments, we define a parametric model of optimism that can be used to quantitatively understand its behavior. There are several component to any such model that must be defined that correspond to instances of optimism in different application domains. These components include (1) the workload model, (2) the optimism control model, and (3) the failure dependency model. We discuss each of these in turn.

3.1 Workload Model

The workload model we use here is shown in Figure 1. This is a simple, asymmetric, pair-wise interaction between two nodes, say P_i and P_j. Optimistic computation at each node is comprised of $1 \leq k \leq K$ processing blocks, which

determines the length of the entire computation. On node P_i, at the end of $blk_{i,k}$, a message $msg_{i,j,k}$ is sent. This message requires some amount of processing time, say $msgblk_{i,j,k}$, after which a short *success* or *failure* reply message is sent. In the absence of other useful work (or optimistic execution) P_i would have to idly wait until $reply_{i,j,k}$ is received. Only then could $blk_{i,k+1}$ begin. This defines the following basic parameters for our model of optimism:

- Number of processing blocks,
- Block processing time,
- Message length, and
- Message Block processing time

Since the communication between P_i and P_j will be through a network, we also have the parameters of

- Communication Bandwidth, and
- Communication Latency.

While some of these parameters are straight-forward, others can be characterized or represented in multiple ways. Clearly the number of processing blocks, and perhaps the message length, can be represented by a static value. In different applications, however, the block processing time and message block processing time could be variable; in which case, it would be more realistic to represent the parameter by a mean with some distribution, rather than a simple, static value. Similarly, in practical situations using shared network resources, the communication bandwidth and latency will be variable. Much work has been done trying to characterize network bandwidth and latency for performance prediction reasons, but this is out of scope for this paper. Our primary goal for this paper is to define the model and make preliminary investigations into the overall behavior of optimistic computation in grid environments.

3.2 Optimism Control Model

For the purposes of this parametric study, the optimism control model refers to two key aspects of optimistic computation. Any block message sent to another node may succeed or fail. Hence, there is a *success probability* associated with block messages. For an application that is executing optimistically, this probability is a result of the application's own behavior. For modeling purposes, however, we will use the *success probability* as an independent variable, and measure its effect on workload execution. How this should be characterized is also application-dependent, but again, it could be a simple static value, a distribution, a trace of values, or whatever representation is most feasible and effective.

Furthermore, the very concept of optimistic computation means that computation proceeds optimistically without waiting for some result, or in this case, a return message. In our model of optimistic computation, this means that more than one block messages could be outstanding at any given time. We refer to the number of such outstanding messages as the *level of optimism*. Hence, an optimism level of 1 actually denotes *non-optimism*; that is to say, for any host-pair interaction, there is only one outstanding block message at a time, and the sending host waits until a reply returns. An optimism level of 2 allows a second block message to be sent before waiting for a reply to the first block message, and so on.

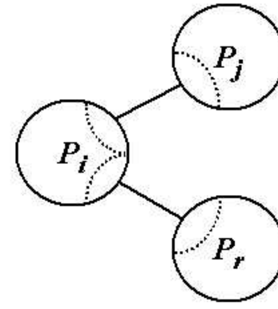


Figure 2. Node with multiple interfaces.

3.3 Failure Dependency Model

Clearly when an optimistic block succeeds, the computation may proceed. When it fails, however, what happens? This is defined by the failure dependency model.

In general, the computation at P_i must *roll-back* to some *previous state*. Not only is there a processing overhead associated with rolling back, it means that some segment of the computation must then be redone. Roll back is typically accomplished by methods for *incremental state saving*. What state must be saved and how complicated it is to do this incrementally is application-dependent.

Furthermore, we must realize that while the workload model is defined to be a basic, pair-wise interaction, one particular node may interact with multiple other nodes optimistically, as illustrated in Figure 2. Node P_i has optimistic interactions with both P_j and P_r that will determine the progress on its overall workload. The dependency model must capture if there are dependencies among the set of pair-wise interactions. That is to say, it must capture if a failure on the i, j interface can affect progress on work associated with the i, r interface. An i, j failure must have some level of roll-back on the workload associated with that interface, but it may or may not cause a roll-back on the i, r workload.

4 Experimental Approach

With this model of optimistic computation, we proceeded to build a workload generator that produced sets of fundamental, pair-wise interactions in parallel. We then evaluated these workloads in an environment where the network topology, bandwidths and latencies could all be controlled to emulate a wide-area grid. This enabled us to run set of experiments across the parameter space defined by our model and characterize the behavior of optimistic computation on a grid. We emphasize that this tool is not a simple, event-driven simulator. We are emulating optimistic computation where a genuine parallel implementation is generating real network traffic.

4.1 The Optimistic Workload Generator

Our workload generator was a parallel code written in MPI using MPICH-2 1.0.3 and TCP/IP. This code was structured as a state machine that would service events on any interfaces without blocking. For our purposes, events were the end of a "processing era" or an incoming message. Processing eras were timed against the system clock with compensation being made for interleaved operations, e.g., receiving a message.

On start-up, this code would initialize from a configuration file that defined the number of nodes, which nodes had pair-wise communication, and the parameters of optimism on each interface. To summarize, the following parameters were defined for the workload generator:

- Number of processing blocks,
- Block processing time,
- Message length,
- Message Block processing time,
- Communication Bandwidth,
- Communication Latency,
- Level of Optimism, and
- Success Probability

Note that the number of processing blocks defined the length of any computation. This is the only parameter that is the same across all nodes and interfaces in a computation. All other parameters could be set on a per interface basis, or to a default value.

With so many parameters, it is a challenge to design feasible experiments that can capture and demonstrate their effect on the performance of optimistic computation. For this reason, we have used a number of simplifying assumptions.

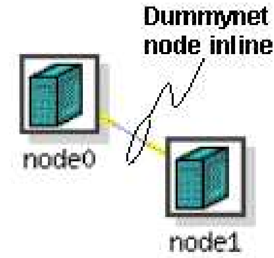


Figure 3. Simple pair topology.

While arguments could be made that more realistic assumptions could be used, we argue that the assumptions we make here are adequate for capturing the fundamental behavior of optimistic computing in grid environments. In this paper, the goal is just to quantify the first-order behavior of optimistic grid computing.

That said, for any one computation, we adopted simple static values for all parameters: the block processing time, message length, block message processing time, bandwidth, latency, success probability, and level of optimism. We also adopted a very simple failure dependency model. When a block message failed, it only caused the node to roll-back to the failed block and restart the computation from that point (implicitly causing any other outstanding block messages to be redone). We did not explicitly model the overhead of any state saving. Furthermore, we assumed that there was no interaction between interfaces on any one node. That is to say, failure on one interface did not cause any roll-back on other interfaces.

Again, while the argument could be made that these assumptions are over-simplifications, these assumptions are the first step in evaluating optimistic grid computing through emulation. As we shall see, the data produced by these initial experiments is considerable, even with these simplifying assumptions. Refinement of these experiments to address other assumptions must be left to future work.

4.2 The EmuLab Platform

All experiments were run on an EmuLab system. EmuLab was originally developed at the University of Utah under NSF sponsorship to provide a network emulation facility [22]. EmuLab is essentially a specialized cluster connected by a configurable switch. Through a web-based portal, users can define experiments which entails defining a *topology* of nodes. For each node, the entire software environment can be specified. This includes which operating system will run on the node, and all installed libraries and software packages. For this purpose, entire *system images* can be built. The network behavior on links between nodes can also be controlled. Dummynet nodes [15] can be inserted on links

whereby the apparent bandwidth and latency experienced on a given link for all traffic can be set.

When an experiment is actually instantiated, a set of nodes is dedicated to the experiment, the physical routes among them are configured in the switch, and the appropriate system images are booted. At this point, the user can log-on to any node in the experiment and work as usual. Of course, this typically means running scripts and codes to collect experimental results.

The EmuLab system at The Aerospace Corporation (which is affectionately called *AeroLab*) currently consists 32 2.4-GHz. Xeon processors with 512MB of RAM and 80GB of disk. All nodes have multiple Gigabit Ethernet network interfaces. AeroLab has one Cisco 6509 switches, which functions as the *testbed backplane* or *programmable patch panel*. Several other servers provide user, file, web, and serial line service.

For our experiments, FreeBSD 4.7 was run on all nodes, including the Dummynet nodes. We also note that the TCP send and receive buffers were set to 16MB in all cases.

4.3 The Experiments

For this paper, we only have space to report on the simple pair topology shown in Figure 3, which had a Dummynet node on the single link. On this topology, *symmetric* experiments were run where both `node0` and `node1` generated block messages and responded to each other. Clearly running the workload generator on both nodes increases the bandwidth demand on the network and the processing load on each node, with the processing of blocks and messages being interleaved.

5 Experimental Results

We now present the experimental results. These experiments ran between 50 seconds, with small messages, high bandwidth, low latency, high success probability, etc., and almost 8600 seconds, with large messages, low bandwidth, and high latency, low success probability, etc. Again for reasons of space, we will only present the *normalized performance* where the optimistic performance in a "distributed environment" is normalized against the best possible non-optimistic performance in a "local environment. That is to say, for example, the optimistic performance with 100 msec. of additional latency will be compared to the non-optimistic performance with zero additional latency (where all other parameters are held equal). This normalized comparison will give us an understanding of how well optimistic execution can compensate for a distributed environment and produce performance closer to that of a local environment. Hence, normalized performance close to 1 means that optimistic execution in a distributed environment was able to produce performance close to that of non-optimistic execution in a local environment.

5.1 Latency-Success Probability Experiments

Figure 4 presents the normalized performance for this topology essentially across four parameters. Nine graphs are presented in a two-dimensional matrix. From left to right, each column contains the results from Optimism Levels 2, 3, and 4. In each column, from top to bottom, the block processing and message processing times are set to 100, 200, and 400 msec. (In each individual graph, the block processing time and message processing time are equal.) In all nine graphs, the message length was 1000 bytes, and the network bandwidth was 1Gb/sec. In all graphs, 500 blocks were computed. This produced experiments long enough to generate useful measurements but without being excessively time-consuming. Each graph shows the relative performance for six different Success Probabilities (0.50, 0.65, 0.80, 0.90, 0.95, and 0.99) as a function of six network latencies (0, 20, 50, 100, 200, and 400 msec.). These network latencies are *one-way*; not round-trip. (A one-way network latency of 400 msec. may seem huge, but AeroLab's Dummynet nodes were provisioned to emulate network latencies up to geosynchronous orbit.) With this presentation, we can get an understanding of how four parameters in our model – level of optimism, block/message processing time, latency, and probability of success – affect performance. We will call this the *Latency-Success Probability* experiment.

This normalized performance is shown in Figure 4 and provides an important insight. For the 100 msec. processing time case (top row), optimism (with a good probability of success) can approach the performance of non-optimistic, local computation. However, as additional network latency is added, the normalized performance drops off. As higher levels of optimism are used, though, performance improves and is closer to that of local execution. (We see that even through Opt Level 4 with a network latency of 400 msec. can produce a speed-up of 4x over non-optimistic execution in a distributed environment, this is not enough to completely compensate for the network latency and normalized performance is still below 80%.) When the processing times are higher, and closer to that of the network latency (bottom row), we see that the normalized performance primarily depends on the success of probability. Even though lower speed-ups are realized, it is enough to approach the performance of local execution – just so long as the probability of success remains high.

5.2 Bandwidth-Message Length Experiments

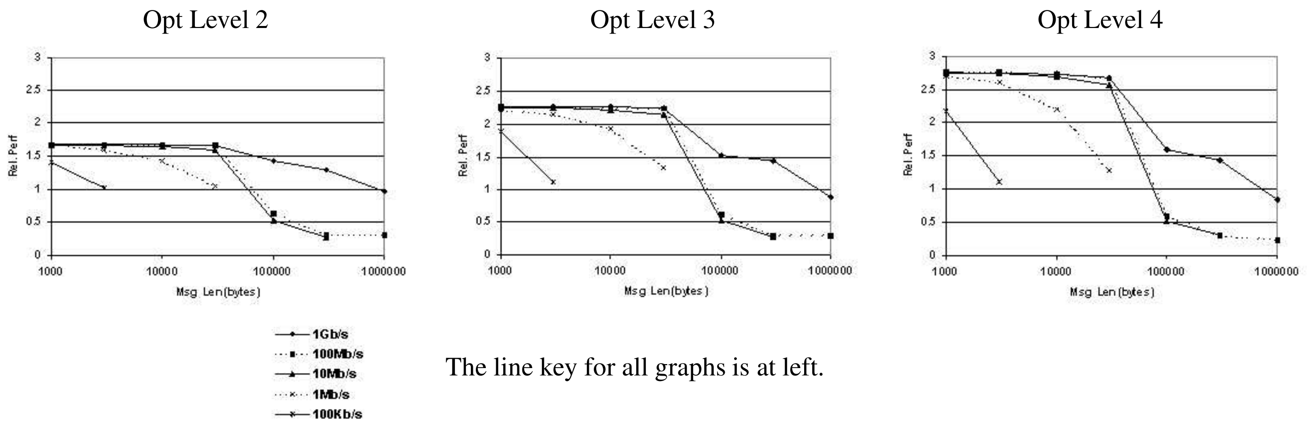
The experiments so far have focused on the effects of latency and success probability on optimistic speed-up. We

Opt Level 2 | Opt Level 3 | Opt Level 4

Prob 99
Prob 95
Prob 90
Prob 80
Prob 65
Prob 50

The line key for all graphs is at left. In each graph, the Processing and Message-Processing Times are equal. In each column, from top to bottom, these times are 100, 200, and 400 msec.

Figure 4. Normalized performance for the Symmetric Pair, Latency-Success Probability experiments.



The line key for all graphs is at left.

Figure 5. Normalized performance of the Symmetric Pair, Bandwidth-Message Length experiments.

now look at the effects of message size and bandwidth in the symmetric pair topology. For these experiments, we fix the block processing and message processing times at 50 msec. and the network latency at 200 msec. (This will allow optimistic execution to demonstrate a broader range of effect.) The success probability is fixed at 0.90. Again, 500 processing blocks were computed. Here, message lengths and network bandwidth are varied. Message lengths of 1 KB, 3 KB, 10 KB, 30 KB, 100 KB, 300 KB and 1 MB are used. Network bandwidths of 100 Kb/s, 1 Mb/s, 10 Mb/s, 100 Mb/s and 1 Gb/s are used. Given the range of message lengths and bandwidths used, we did not run all combinations – sending 1 MB messages over a 100 Kb/s link would take just too long. Hence, we only ran messages up to 3 KB over 100 Kb/s links, messages up to 30 KB over 1 Mb/s links, messages up to 300 KB over 10 Mb/s links, and messages up to 1 MB over 100 Mb/s and 1 G/s links.

Figure 5 shows the performance results normalized to that in a "local" environment, i.e., the performance using a bandwidth of 1 Gb/sec. Here we see that increasing the level of optimism increases the general relative performance. This general performance does not, however, approach the level of optimism since there are less spare processing cycles on each node. We also see that for the larger messages and higher bandwidths, the severe bottleneck in performance begins with messages of 100 KB. This should not be surprising since increasing the level of optimism also increases the bandwidth demand, thereby exacerbating the drop-off in performance seen in the non-optimistic performance for messages larger than 100 KB. What is also pronounced is the drop-off in performance for the slower bandwidths, e.g., 100 Kb/sec. and 1 Mb/sec. Here, however, the bandwidth demand is most likely exceeding the limited available bandwidth.

Hence, as long as the middleware and the network can meet the bandwidth demand, optimism can actually produce significant speed-ups. As soon as the network bandwidth is saturated, however, performance sinks precipitously – in the worst cases, below the non-optimistic performance.

6 Discussion and Conclusions

The goal of this work was to shed some light, or bound the discussion, on how effective optimistic computation can be in distributed environments and under what conditions – and not just to understand this particular model of optimism or the idiosyncrasies of this particular implementation. That said, the results provided by this model of optimism, workload generator and EmuLab experiments are both instructive, useful and problematic at the same time. We can clearly make a number of qualitative observations.

In these experiments, we see the effect of processing times, latency, and available cycles on a host. If the block and message processing times are large compared to the message latency, then optimism is less beneficial. Also, if a host has a heavier processing load, such as in the square topology where each node symmetrically interacts on two interfaces, then optimism is less beneficial unless latencies are much higher. That is to say, optimism is most beneficial when the network latency dominates in the processing cycle.

This may not be surprising but these results do serve to confirm our qualitative intuitions about how optimism should work. To summarize in a more concise way, if the processing time requires $1/n$th of the processing cycle (and the rest of the cycle is network latency), then an optimistic speed-up of *up to n* is possible, if the processor and network do not become saturated, and the probability of success is good with a low failure overhead. We have seen this – using a real parallel code, running over a real network protocol. Likewise, we have seen break-even relative performance when the latency is only 1/6th of the processing cycle time with a 90% probability of success that produces 85% of "local" performance. The many factors governing optimistic behavior means that there is a significant range where optimism can exceed break-even performance and provide significant speed-ups.

The next step naturally it would be useful to be able to make *accurate predictive statements* for *specific applications* using optimistic execution: *given the parameter values that characterize its behavior, how much speed-up could be routinely realized?* To be able to make such predictions accurately, however, will require much more work. As we have already noted, a number of assumptions were made to make the experiments presented here feasible. The use of static processing times, message lengths, success probabilities, and also network bandwidth and latency, all contribute to the quantitative results seen. Distributions of values could be used for these variables, and network traces could be "replayed" to produce competing traffic with different results. The failure dependency model was also something we did not investigate thoroughly. It seems unlikely, however, that such refinements would completely eliminate all possible optimistic speed-up, rather it would only move where the break-even points come and the degree to which benefits are seen. Quantifying such differences experimentally is a worthy future endeavor.

Finally we wish to emphasize that the ultimate goal of this work is to enable grid applications to become more latency-tolerant using techniques such as optimistic computation. Clearly this is a hard question that can best be answered by direct experience. Actually evaluating applications such as optimistic mesh generation, however, may involve even more work than reported in this paper. Nonetheless, as grids and service architectures become more commonplace, these evaluations will be done and the fundamental technique of optimistic computation will be put to use.

Acknowledgments

This work was supported by NSF EIA-0203974: Mesh Generation and Optimistic Computation on the Grid, through a subcontract with the College of William and Mary, and by the Aerospace Parallel Computing MOIE project. The author gratefully acknowledges the many conversations with Eric Coe and James Stepanek on the use of AeroLab and its network behavior.

References

[1] Allen, Dramlitsch, Foster, Karonis, Ripeanu, and Seidel. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. In *Supercomputing*, 2001.

[2] A. Back and S. J. Turner. Transformations for the Optimistic Parallel Execution of Object-Oriented Programs. In *International Conference on Parallel Object-Oriented Methods and Applications (POOMA)*, 1996.

[3] R. Bubenik and W. Zwaenepoel. Semantics of optimistic computation. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pages 20–29, Washington, DC, 1990. IEEE Computer Society.

[4] R. Bubenik and W. Zwaenepoel. Optimistic make. *IEEE Trans. on Computers*, 41(2):207–217, February 1992.

[5] N. Chrisochoides, C. Lee, and B. Lowekamp. Mesh generation and optimistic computation on the grid. In V. Getov, M. Gerndt, A. Hoisie, A. Malony, and B. Miller, editors, *Performance Analysis and Grid Computing*, pages 231–250. Kluwer Press, 2003.

[6] C. Cowan. Optimistic Programming in PVM. In *2nd PVM User's Group Meeting*, May 1994.

[7] Crispin Cowan and Hanan Lutfiyya. A wait-free algorithm for optimistic programming: HOPE realized. In *International Conference on Distributed Computing Systems*, pages 484–493, 1996.

[8] Electric Communities. The E Extensions to Java. www.erights.org/history/original-e/datasheet.html.

[9] R. M. Fujimoto. Parallel discrete event simulation. *CACM*, 33:31–53, 1990.

[10] D.R. Jefferson. Virtual Time. *ACM Trans. Prog. Lang. Systems*, 7:405–425, 1985.

[11] H. Lutfiyya and C. Cowan. Optimistic Language Constructs. In *Workshop on Research Issues in the Intersection of Software Engineering and Programming Languages*, 1995. At ICSE-17.

[12] Namprempre, Sussman, and Marzullo. A Framework for Optimistic Execution in CORBA Environment. www-cse.ucsd.edu/ cnamprem/concurrency.ps, 1999.

[13] D. Petrou. *Cluster scheduling for explicitly speculative tasks*. PhD thesis, CMU-PDL-04-112, Dept. of Elect. and Comp. Eng., Carnegie Mellon Univ., 2004.

[14] Martin C. Rinard. Effective Fine-Grain Synchronization for Automatically Parallelized Programs Using Optimistic Synchronization Primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, 1999.

[15] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[16] Jeffrey S. Steinman. The WarpIV Simulation Kernel. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 161–170, Washington, DC, USA, 2005. IEEE Computer Society.

[17] J.S. Steinman, C.A. Lee, L.F. Wilson, and D.M. Nichol. Global virtual time and distributed synchronization. In *IEEE 9th Workshop on Parallel and Distributed Simulation*, pages 139–148, June 14–16 1995. Lake Placid, NY.

[18] Tang, Perumalla, Fujimoto, Karimabadi, Driscoll, and Omelchenko. Optimistic simulations of physical systems using reverse computation. *Simulation*, 82(1):61–73, 2006.

[19] Stephen J. Turner. Optimistic Network Computing and its Performance Control. In *International Conference on Supercomputing, Workshop on Performance Data Mining*, 1997.

[20] Wallach, Hsieh, Johnson, Kaashock, and Weihl. Optimistics active messages: A mechanism for scheduling communication and computation. In *PPoPP*, 1995.

[21] M. Weiner and Vijay Garg. Optimistic Distributed Computation via Code Undo. ECE Parallel and Distributed Systems Lab TR-PDS-2004-007, University of Texas, 2004. Available at maple.ece.utexas.edu.

[22] White et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symp. on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002.