Dynamic Mapping of Activation Trees
Thesis Proposal

Peter A. Dinda

January 21, 1998

Abstract

An interactive application responds to aperiodic user input with computation that can be expressed as activation trees. By bounding the execution time of these trees, we can improve the responsiveness of the application. The application runs in a typical computing environment: a group of uncoordinated but measurable hosts interconnected with a measurable local area network. The environment serves a number of independent users and we do not control it. However, using a remote execution mechanism, we can execute each node (procedure call) of a tree on any host on the network. We want to use this freedom by dynamically mapping nodes to hosts in order to maximize the percentage of trees that execute within their allotted time.

My thesis is that this dynamic mapping problem can be solved using history-based prediction. In essence, the idea is to monitor the system and the application and use the measurements to predict how their behaviors will evolve. Each node is then mapped to the host which best serves to match the predicted future application behavior to the predicted future system behavior.

We present two strong pieces of evidence that argue that this is an effective approach. The first is an extensive analysis of host load traces on a wide variety of host systems which strongly suggests that load can be predicted from its history. The second is a history-based prediction algorithm that achieves near optimal performance for a simplified variant of the problem.

For my thesis, I will extend these results to develop a history-based prediction algorithm for the full dynamic mapping problem. The algorithm will be developed using a realistic trace-driven simulation environment based on host load traces, network traces, and traces of activation trees collected from real applications. Because of its highly realistic nature, I intend to also evaluate the algorithm using the simulator and a set of benchmarks. In addition, I will incorporate my algorithm into a distributed object system to show that it can work in a real system.

1 Introduction

Users demand responsiveness from interactive applications such as image editors, scientific visualization tools, modelling tools based on physical simulations, and games. Such applications react to aperiodically arriving messages that arise from user actions. In response to each message, the application program executes a message handler, the call to which forms the root node of an activation tree that unfolds as computation progresses. The computation represented by the tree creates visual and aural feedback for the user. This feedback helps determine the subsequent actions of the user. The aperiodicity in message arrival is a result of having a "human in the control loop." To be responsive, the application must execute the activation tree induced by each message as quickly as the user reasonably expects.

At one point, creating applications that were responsive in this sense was relatively easy since the user's machine was very predictable from the point of view of the programmer. Today, however, the user's machine is becoming increasingly less predictable due to operating system jobs, daemons, other users' jobs, and the like. These external factors make the actual computation rate the programmer can expect vary in complex ways. Further, the user's machine may simply not be fast enough or have enough memory to perform some computations responsively. On the other hand, the user's machine is no longer alone — there are many other hosts on the local area network which it can talk to. As the overheads of remote execution facilities such as remote procedure call (RPC) systems, object request brokers (ORBs), and distributed shared memory (DSM) systems decline, it becomes more tempting to execute smaller chunks of work remotely to achieve the responsiveness that interactive applications require. Unfortunately, the overwhelming majority of networks and hosts do not provide any sort of resource reservations, or even priority-based scheduling that a programmer could build upon to make a group of hosts behave in a more predictable fashion than an individual host. However, these environments provide increasingly sophisticated measurement tools.

A best effort real-time service would greatly simplify building interactive applications in this context. Using such a service, the programmer would specify time bounds for executing the activation tree rooted at a procedure call and the service would attempt to meet the bounds by measuring the system and using the remote execution facility. The design space for such a service is vast. My approach is to dynamically map the nodes of the activation tree to appropriate hosts as the tree unfolds during execution, making each mapping decision in pursuit of the goal of achieving the overall bounds on the tree.

The core problem that needs to be solved to make such a service possible is a dynamic mapping problem: We are given a stream of aperiodically arriving activation trees. Each tree has an associated upper and lower bound on its execution time and its structure becomes known only as we sequentially execute it. We can execute an activation tree node on any one of a group of uncoordinated but measurable hosts interconnected with a measurable network. The hosts and network can all have unrelated computation and communication traffic on them. Under these conditions how do we map the nodes of the trees to the hosts so that the percentage of trees that execute within their desired bounds is maximized?

My thesis is that this dynamic mapping problem can be solved using history-based prediction. History-based prediction involves predicting the future values of a series using a window of previous values. In essence, the idea is to monitor the system and the application and use our measurements to predict how their behaviors will evolve. Each node is then mapped to the host which best serves to match the predicted future application behavior to the predicted future system behavior. By adapting our mapping strategy as the activation tree is traversed, we can better explore the system and handle erroneous predictions.

There is strong evidence that history-based prediction is an effective way to approach this dynamic mapping problem. The evidence comes from two distinct threads of investigation. The first thread involves an extensive analysis of load traces collected on a large number of hosts from four common classes (experimental and production compute clusters, desktop workstations, and compute servers) on which we expect to run our interactive applications. The analysis strongly suggests that load is predictable from its history. The second thread involves using history-based prediction of execution times to achieve near optimal performance for a simplified variant of the dynamic mapping problem.

Extensive analysis of host load traces reveals three properties which strongly suggest that host load is predictable from past load measurements. These properties are self-similarity, epochal behavior of local frequency content, and the phase space behavior of the load. Self-similarity implies a long-memory process where the future depends on a long stretch of the past in a possibly complex way. By epochal behavior of the local frequency content, we mean that the manner in which the load varies (the frequency content of the load signal) is relatively stable for long periods of time (epochs.) Because each epoch is long, we have an extended opportunity to understand the behavior within it. The third property is that the phase space behavior of the load is relatively simple, and thus easier to predict.

I have developed a fast, simple, and near optimal history-based prediction algorithm for a simplified variant of the dynamic mapping problem and evaluated it through extensive, realistic simulation based on the load traces. In the simplified problem only leaf nodes are mapped and communication costs are ignored. There are several conclusions to draw from the results. First, dynamic mapping can make a significant difference in the percentage of calls that meet their deadlines - in many cases the near-optimal algorithm allows almost 100% of calls to meet their deadlines while random or host-specific mappings barely manage 30-40% of the calls. The second conclusion to draw from my experience is that developing a good history-based prediction algorithm for even the simplified problem is a non-trivial undertaking. Simple algorithms behave badly, while complex generic approaches such as neural networks work well, but have overheads that are far too high. However, there is a middle ground — effective yet efficient algorithms do exist. The final conclusion is that there remains room for improvement.

For my thesis, I will extend these two threads of investigation to develop a history-based prediction algorithm for the full dynamic mapping problem. The algorithm will be developed using a realistic trace-driven simulation environment based on host load traces, network traces, and traces of activation trees collected from real applications. Because of its highly realistic nature, I intend to also evaluate the algorithm using the simulator and a set of benchmarks. In addition, I will incorporate my algorithm into a lightweight distributed object system to show that it can work in a real system.

We begin this proposal in Section 2, which explains our model of interactive applications and responsiveness in detail. Section 3 describes our machine model and the assumptions we make about available services. We describe our execution model for activation trees in Section 4. This leads to Section 5, which describes the dynamic mapping problem, and Section 6, which describes our history-based prediction approach to the problem and contrasts it with other approaches. The next two sections present our preliminary results. In Section 7 we describe our analysis of host load traces, while in Section 8 we discuss our results for the simplified dynamic mapping problem. Section 9 outlines the thesis work and Section 10 describes the expected contributions. Finally, Section 11 differentiates this thesis from related work.

2 Application model

Our model interactive application has a very simple main loop which repeatedly gets the next message from the operating system and calls the appropriate handler for the message. Messages result from user actions that occur aperiodically. Message handlers compute display and sound updates that serve as feedback to the user and determine his subsequent actions. The call to the message handler is the root node of an activation tree whose structure is not fully known until after it has been executed, but which depends on the handler, the message, and the state of the program. It is important to bound the execution time of the tree from above in order to maintain responsiveness and from below since it need not execute faster than humans can detect or expect. Even for trees which cannot possibly be executed with imperceptible delay, bounding their execution times makes the user's waiting time predictable and consistent.

Example applications

We will use two interactive applications as examples in the following. The first is an image editing application and the second is a acoustic room modeling application. [1] These applications serve as models for broader classes of applications. The room modeling application is a computer aided design application based on a physical simulation and can also be seen as a computational steering application. Image editing is typical of the large document editing applications common on personal computers. As described in Section 9, we intend to extend our application set using an infrastructure to collect relevant traces from off-the-shelf programs. Ideally, we will be able to add existing commercial Microsoft Windows applications to our application set.

Image editing gives the user tools to manipulate an in-memory visual image as a whole and in part. Some tools involve image-processing operations such as boxcar convolution on large regions of the image, while others involve emulating real-world tools such as pens, brushes, or spray paint cans. It is important to point out that image sizes are rapidly growing, and the limits imposed by photographic film, drum scanners, and digital cameras imply that truly vast (100s of megabytes to gigabytes) images will need to be edited. At the same time, the functionality of image editing software, in terms of the sophistication of image filters and how they can be applied is rapidly advancing.

Acoustic room modeling involves designing a human space using a CAD tool and being able to hear what such a room will sound like from different positions in it. For a given room configuration (room geometry and material properties, listener positions, and loudspeaker positions), we compute an impulse response function for each listener/loudspeaker pair. For a particular listener/loudspeaker pair, we convolve the music signal coming from the loudspeaker with the impulse response function, giving the room-filtered signal the listener would hear from that loudspeaker. By doing this for each loudspeaker and summing the resulting room-filtered signals, we simulate what the listener would hear given the room configuration. [2] The impulse responses are computed by simulating the wave equation using a finite difference method [71]. In steady state, periodic convolution and summation occurs to compute the sound output. When the user changes the room configuration by moving a loudspeaker, wall, or himself, the (expensive) computation of the impulse responses is repeated.

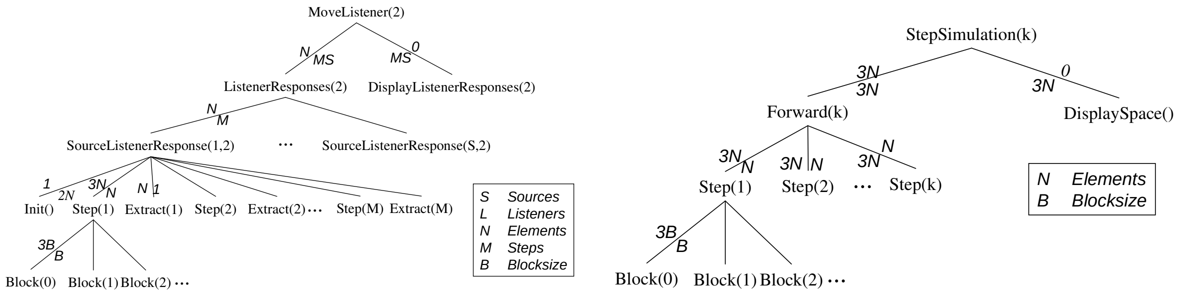
As an alternative to sound, the user can view the impulse response functions directly, or can view the frequency response characteristics of the room computed from them. In this mode, the acoustic room modeling application shares many characteristics with the design optimization applications (for example, shape optimization [50]) currently being discussed in the Quake project [5, 4]. In both acoustic room modeling and design optimization, the user repeatedly adjusts the model parameters (furniture position and composition; soil and rock composition), simulates the physical system, and views the results. In the room modeling application, the user iteratively adjusts the parameters to achieve a flat response, while in the Quake case, the user is iteratively debugging his physical model of geological structures and composition. In both cases, the user may also single-step the simulation to discover what is happening in detail.

Aperiodic user actions generate aperiodic messages

Most computation in an interactive application arises from aperiodic user actions that result in aperiodic messages being delivered to the program. The aperiodicity is due to from the variable "think time" humans need to decide their next action [28]. Consider a user using a brush tool to draw freehand in an image editing application. Every time he moves the mouse, the operating system sends the application a message

[1] To better understand these applications, we implemented very simple versions which we believe are similar to their full-featured counterparts (eg, Adobe Photoshop [1]) except for their feature sets.

[2] Ignoring the listener's Head Related Transfer Function, Doppler effects during movement, and other issues that are beyond the scope of this document. Interested parties should look at [8].



(a) Listener Moved (b) Step Simulation

Figure 1: Activation trees from acoustic room modeling application

announcing that fact. He may slow down due to indecision, or stop, or release the mouse button ending his drawing. Next, he may select a menu option to run a boxcar convolution on the whole image resulting in a menu message that delivers the news, or he may pause for a second or an hour. The aperiodicity is even more overt in the acoustic room modeling application, where the user is, in effect, meandering around the simulated room.

Message handlers and activation trees

In response to the arrival of a message, the application calls the appropriate handler for the message. The handler performs the necessary computation, which may be very simple or very complex. For example, a message announcing mouse movement in the acoustic room modelling application may simply mean that a line representing a room wall needs to be moved, or it may require that all the impulse response functions be recomputed.

The call to the message handler is the root node of an activation tree. The structure of the tree depends on the message and program state. In general, the structure is unknown until after it has been executed. Figure 1 shows two activation trees from the acoustic room modeling program. To the left of each edge is the amount of data the called procedure needs, while to the right is the amount of data it produces. In Figure 1(a) a listener has changed his position, and thus the impulse responses from each of the sound sources to that listener must be recomputed. For each source/listener pair, the simulation is stepped a sufficient number of times to propagate the initialize impulse to the listener, including several reflections. The series of pressure values at the listener position is extracted to form the impulse response, which is then displayed. In Figure 1(b), the underlying simulation itself is being stepped and visualized to aid in debugging. After every k steps, the grid of all the pressure values in the room is displayed.

Feedback

The message handler ultimately generates feedback to the user in the form of display updates or sound. This feedback helps the user determine what his next action will be and when he will undertake it. On a large time scale this is clear - the user of an image editor will not apply another filter until he has seen the effects of the one he just applied. On smaller time scales it can be even more important. Consider the same user drawing freehand. This results in a stream of mouse movement messages, and the handler changes a few pixels for each message. Without these few changed pixels, the user is drawing blind.

Responsiveness

Achieving responsiveness in an interactive application amounts to providing timely but not necessarily too timely feedback to individual user actions. If the feedback arrives too late or there is too much jitter for a series of similar actions, the utility of the program is degraded, perhaps severely. For example, high-jitter or long latency feedback makes freehand drawing impossible in an image editor.

Although we require fast feedback, there is no benefit to it arriving too early, since humans can only perceive rapidity up to a point. It makes no difference to the user of an image editor whether it responds to his freehand drawing in 5 milliseconds or 50, but the difference between 50 and 150 is the difference between usability and uselessness.

Beyond this threshold of human perceptability, there other similar thresholds of annoyance in waiting time and "sweet spots" between them. To the user, responsiveness means knowing in which sweet spot the wait for an action will be and that the wait for similar actions will always be in the same sweet spot (jitter will be low.) Intuitively, the user is annoyed at having to wait, and so wants to do something with that time. When the waiting time crosses a threshold of annoyance, he decides to do something else during the wait. If he knows which sweet spot the wait is in, he can meaningfully decide what to do during the wait. For example, you may wait and read a magazine during a 10-30 minute oil change, but you'll probably leave your car and go to work during a 100-300 minute tune-up. If the mechanic tells you it will take 10-30 minutes, but it takes 100-300, you will be very annoyed. Conversely, if he says 100-300 and it really takes 10-30, you will get no benefit.

Similarly, applying a filter to a realistic sized image or recomputing impulse response functions in a reasonable sized room will necessarily take a perceptible amount of time, but the difference between waiting 10-30 seconds or 100-300 seconds is the difference between waiting patiently and getting a cup of coffee. Another example is loading a web page from a remote site which again necessarily takes a perceptible amount of time, yet still has a sweet spot.

What we really want is to bound the execution time of a call to a handler from above and below in order to fit it into the sweet spot the user or programmer expects. Of course, if we achieve the upper bound, the lower bound can always be achieved by inserting artificial delay. However, note that minimizing the actual execution time and then adding delay is much more difficult than simply trying to meet the bounds.

Tradeoffs and discussion

Stepping the acoustic room modeling simulation by k steps, as shown in Figure 1(b), is a good example to illustrate the tradeoffs involved in remote execution of calls. Consider a system with two hosts: A, which is the user's workstation and computes at 25 MFLOP/s, and B, which is a shared compute server which computes at 50 MFLOP/s. The two machines are connected with an 10 MB/s network. We begin executing on A and have to decide whether to execute Forward(k) on either A or on B. Let us suppose for the moment that the whole subtree rooted at Forward(k) will execute on the same host as the root node. The problem size N depends on the size of the space to be simulated and the peak frequency we want to resolve. Omitting the derivation for space reasons, for a cube 4 meters on a side and a frequency of 3000 Hz (telephone quality), the problem size is $N = 2,744,000$, where each of the N values which grid the cube is a double (8 bytes) representing the pressure at a point in the space. For the purposes of this discussion, we also associate one scalar material property with each point.

A simulation step involves computing the tth set of N output values from the $t-1$th and $t-2$th sets, and the material properties set using 26 floating point operations per point. On A, a step can be computed in $26N/25$ MFLOP/s $= 2.86$ seconds, while on B computing a step takes only $26N/50$ MFLOP/s $= 1.43$ seconds, but we must first transfer $3N$ doubles (the $t-1$ and $t-2$ pressure values, and the material property values) to B and then transfer $3N$ doubles (the last three pressure value sets) back to A to be displayed.

This requires $8 \times 6N/10$ MB/s $= 13.18$ seconds. If the number of steps k is less than $\lceil 13.18/1.43 \rceil = 10$, it is faster to perform the computation on A, otherwise it is faster to do it on B. Notice, that when stepping the simulation for debugging, k is likely to be small, perhaps even one. On the other hand, to recompute the impulse responses due to a listener moving (Figure 1(a)), we need to perform at least $N^{1/3} = 140$ steps, and probably several times that number in order to capture several reflections.

Since B is a shared machine, we may not always get the full 50 MFLOP/s from it, so we must take the future load on B into account when we make our decision. For example, if other jobs put a unity load on B, then there is no tradeoff point. Similarly, the actual bandwidth available on the network may vary, again changing the tradeoff point. It is also important to point out that with a large number of steps, an incorrect decision on where to execute Forward(k) or SourceListenerResponse(i,j) will be felt for a long time. However, if we allow ourselves to map individual Step()s (or Block()s in an more fine grain environment), we can potentially better recover from bad decisions.

3 Machine model

Our machine model corresponds to the real world computing environments that most people have access to. In particular, we assume host computers interconnected by a local area network. The host computers have no centralized scheduler or coordinated scheduling mechanism and their local schedulers do not support reservations or real-time scheduling. Similarly, the network does not support any sort of reservation scheme. The hosts execute independent tasks that generate traffic on the network. This is, of course, a description of any modern group of workstations or PCs.

In addition to these commonplace features, we also assume the existence of two extensions. The first is a remote execution facility that makes it possible to seamlessly execute any procedure in the program on any host. The second extension we assume is a measurement facility. The core requirement is that we must have access to a real-time clock on each host that is precise down to the overhead time of the remote execution facility (1 ms for most current systems, conceivably as low as 1 μs for future systems.) Currently, we expect only to be able to accurately measure the execution times of remote procedure calls. However, it may also be useful to be able to measure the constituents of that execution time, such as host load and available network bandwidth. Ideally, it would be possible to measure load and network traffic at the clock granularity and make this information available to every host in the system. However, we also require that any such measurement system should not significantly perturb the system. In terms of performance, it should not use more than a small, known fraction (5%, for example) of available cycles and communication bandwidth, and should not exhibit any feedback effects that would falsely inflate or deflate its measurements.

4 Activation tree execution model

As we discussed in Section 2, interactive applications are characterized by aperiodically arriving messages that induce activation trees. We assume that each tree is of finite size and is executed in strictly depth-first order — each node must return only to its parent. This means that setjmp/longjump and exception-handling mechanisms that behave in similar ways are not permitted. Further, each call (node) must be expressible in terms of data movement to the selected call site, local execution on this data, which may include further calls, and data movement away from the call site. Data movement includes not only the arguments to the call, but also any static data or object state that the call will read or modify. However, any call that requires access to data that may change due to external forces, such as the memory-mapped registers of a device controller, is not allowed.

Each activation tree is executed sequentially according to a depth-first traversal. Using the remote execution facility, we can execute each node of the tree on any host on the system — at every procedure call, the

Symbol	Description
t_{now}	Wall-clock time at which we begin to map the node
t_{min}	Lower bound — Minimum time the node should take to execute
t_{max}	Upper bound — Maximum time the node should take to execute
t_{slack}	Time remaining before $t_{now} + t_{max}$ after executing the node
t_{exec}	Total time to execute the node and its subtree
t_{map}	Time to map the node to a host
t_{incomm}	Time to send the arguments and other data to the host
$t_{compute}$	Time to execute the node on the host and to execute its children
$t_{computelocal}$	Time to perform all of the node's local computation
$t_{execchildren}$	Time to execute all of the node's children
$t_{outcomm}$	Time to return results to the caller
t_{update}	Time to update the mapping algorithm
$t_{nominal}$	Time to execute the node on slowest host with no load
β	Factor by which a host is faster than the slowest host
H	Hurst parameter
$\langle l_i \rangle$	Sequence of host load measurements
$load(t)$	Host load as a function of time

Figure 2: Recurring symbols in this document.

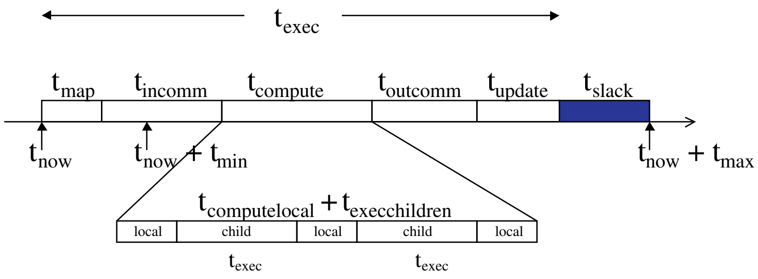


Figure 3: The execution time (t_{exec}) of an activation tree node.

thread of execution can migrate to a different host. Choosing which host shall execute the node is referred to as *mapping* the node, and the decision is made by a *mapping algorithm*. It is important to note that at when we map a node, we are only aware of the portion of the tree that has already been executed.

Of particular interest is the execution time of the activation tree. We define the execution time of a node to be the execution time of the subtree rooted at the node, so the execution time of the tree is simply the execution time of its root node. The symbols we use in the following discussion and elsewhere in this document are summarized in Figure 2. As illustrated in Figure 3, the time to execute a node, t_{exec}, is the time to execute the subtree rooted at the node and consists of the time to map the node (t_{map}), transfer data to the host it will execute on (t_{incomm}), actually compute the node and its children ($t_{compute}$), transfer data back to the caller ($t_{outcomm}$), and update the mapping algorithm (t_{update}). The node's computation time, $t_{compute}$, consists of alternating local computation and execution of child nodes. The sum of local computation times is $t_{computelocal}$. $t_{computelocal}$ depends on the host on which the node is executing and on the amount of work that is to be done, which we capture as the nominal compute time, $t_{nominal}$. Intuitively, $t_{nominal}$ is the time it would take to execute the node on the slowest host in the system under zero load conditions. The sum of child node execution times is $t_{execchildren}$. Each child node execution time is modeled precisely as its parent's. For a leaf node, there is only local computation, so $t_{compute} = t_{computelocal}$.

5 The dynamic mapping problem

In order to be responsive (Section 2), each individual application tree generated during a run of an interactive application should be executed as quickly as the user expects, but there is no benefit to executing the tree too quickly. As illustrated in Figure 3, we formalize this as lower and upper bounds, t_{min} and t_{max}, on the execution time of the tree. The application developer supplies these bounds at time t_{now} as part of the mapping request

`MAP procedure(inargs,outargs,inoutargs,state) IN` $[t_{min}, t_{max}]$

which asks us to execute the activation tree rooted at the call to `procedure` such that $t_{now} + t_{min} \leq t_{exec} \leq t_{now} + t_{max}$, where t_{exec} is the execution time of the tree. Our performance metric is the percentage of such mapping requests that meet their bounds. We desire a mapping algorithm that maximizes this metric.

The dynamic mapping problem induced by this specification is as follows: We are given a stream of aperiodically arriving activation trees. Each tree has an associated upper and lower bound on its execution time and its structure becomes known only as we sequentially execute it. We can execute an activation tree node on any one of a group of uncoordinated but measurable hosts interconnected with a measurable network. The hosts and network can all have unrelated computation and communication traffic on them. Under these conditions how do we map the nodes of the trees to the hosts so that the percentage of trees that execute within their desired bounds is maximized?

6 Approach: History-based prediction with adaptation

The challenge in the dynamic mapping problem is to match future application behavior (the unexecuted portion of the activation tree) to future system behavior (host and network availability.) Recall that the purpose of mapping an individual node is to help achieve the bounds placed on the tree of which it is a part. Unfortunately, when the node is mapped, only the part of the tree preceding it in the depth-first traversal is known. However, we cannot be oblivious to the remainder of the tree, since it determines in part how quickly we should try to execute the node. If we execute the node too quickly, we may well waste resources with no benefit to the user. If we execute it too slowly, we may fail to meet the bounds on the tree. The future system behavior is important because it determines the execution time of the node on each of the hosts. Consider a leaf node - its execution time on a particular host depends on the bandwidth and latency the network will be able to provide and on the load on the host during the node's execution.

Our approach to this challenge is to use history-based prediction and to adapt our mapping strategy as we traverse the activation tree. History-based prediction involves predicting the future values of a series using a window of previous values. In essence, we monitor the system and the application and use our measurements to predict how their behaviors will evolve. Each node is then mapped to the host which best serves to match the predicted future application behavior to the predicted future system behavior. As the tree is executed, we adapt the degree to which the mapping strategy explores the system and what prediction algorithms we use to how deep we are in the tree and how far along in the traversal.

Predicting future system behavior according to a history of measurements is relatively easy to think about. Consider host load, for example. Host load is a scalar quantity and the history is simply a series of load measurements. The prediction algorithm is a function that maps these measurements of the past to a prediction of the future. As we shall see in Section 7, this function may be quite complex, but the concept is simple. Similarly, we can imagine mapping a history of bandwidth measurements into a prediction of the bandwidth at some point in the future. The measurements of the system need not be at this low a level. For example, in Section 8, we collect a history of the execution times of leaf nodes on a host and use these probes to predict future execution times on the host. It is this approach that we plan to extend.

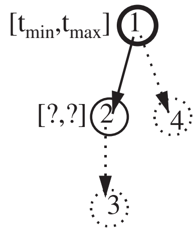


Figure 4: Tree example.

Predicting future application behavior according to a history is a more difficult concept. What does it mean to predict the remainder of the tree when we map a node? What level of detail is required? How do we represent this history and predictions? Our approach is to recursively divide the bounds placed on the root of the tree among the children according to the predicted compute times of the nodes. Given this, we can visualize the prediction problem with only three levels of the tree, as in Figure 4. In the figure, we are executing node 1 (the root node), when it makes a call, making node 2 visible. We must map node 2, but at this point in time, we don't know the execution time of node 2 or about the existence of nodes 3 and 4 (or the subtrees they may be the roots of.) What we want to do is map 2 such that after 2 and 3 have been executed, there is sufficient time remaining in the bound on 1 to execute 4 and the remainder of 1. The question then is: what portion of the bound on 1 should be alloted to 2? A good choice for this bound embodies a sufficiently detailed prediction of remainder of the tree. Furthermore, this representation of the remainder of the tree as a 2-tuple is easy to understand and work with. The prediction algorithm can predict what proportion of the bound on a parent node to allot to a child node node based on a history of such bounds.

Although our current results (Sections 7 and 8) suggest that history-based prediction is an effective approach, it is important to note that it has its limits. We simply cannot measure any system precisely enough to make our predictions always agree with reality. To smooth the differences, we exploit the structure of the activation tree. In particular, we can tune our approach to where we are in the tree and what the overall form of the tree is. Early on in the traversal of the tree, we can be more exploratory in our mapping decisions since if the mapping of a particular node is disappointing, there presumably still remains plenty of time to meet the bounds on the tree. Deep in the tree, as the compute time of a node becomes small and leaf nodes more common, we can switch to a different mapping strategy or prediction algorithm, or simply remain on the current host. For very large trees with high bounds, we may use a different algorithm altogether.

Discussion and comparison to other approaches

The design space of a best effort real-time service is vast, but we believe that our approach represents a very interesting and defensible part of the space. Our approach is based on the belief that prediction algorithms can be developed that perform sufficiently well given only limited, local, application-level measurements. This simplicity would greatly facilitate incorporation into systems such as CORBA. For example, our approach could be incorporated into object references or similar proxies in other systems. No external agent or daemon would be necessary and the service would be available wherever the program could be run. However, there are two important criticisms that can be made. The first is information redundancy. In our approach, each host collects a history of execution times for each procedure on each of the other hosts. This becomes impractical with a large number of procedures or hosts. However, we expect that only a limited number of hosts will be necessary to achieve responsiveness. Similarly, we don't believe that the amount of history we will need to keep for each procedure will be especially large. In the work we describe in Section 8, it is on par with the amount of code in an object reference. The second important criticism is that chaotic behavior could result if several programs using our algorithm were run simultaneously, since they would be oblivious to each other. While a particular algorithm may suffer from this problem, it is not implicit in our approach. Further, since such interaction will likely be incidental instead of purposeful, damping an algorithm's reaction or using randomization would probably be sufficient to reduce the effects of such interaction.

While we compare our approach with dynamic load balancing systems, distributed soft real-time systems, and quality of service systems in Section 11, we will next examine two obvious approaches that are probably on the reader's mind at this point. The first approach, a resource reservation system, is entirely opposite in character from ours. In such a system, bounds would be translated into demands for processor and communication resources (scheduler slots/priority, bandwidth, etc), which the program would then attempt to reserve before executing a node or a whole subtree. If all the reservations were successful, the bounds would be guaranteed to be met. Clearly this determinism has much appeal, but the translation between bounds and resource demands is likely to be quite complex and may require considerable information (hints) from the programmer to get right. Further, over-reserving resources, which is probably a natural reaction, would result in inefficient use of the resources. Finally, to implement a resource reservation system in most computing environments would require operating system and network modifications, which would severely limit the sites and systems where it would be possible.

A middleground approach is one based on a shared measurement system. In such a system, daemons running on each host would regularly exchange measurements of low level quantities such as load and network conditions, reaching a consensus about how the system is behaving as a whole. When mapping a node or a subtree, the local daemon would be consulted and a mapping decision made on the basis of the consensus. For example, the node could be mapped to the host with the lowest load. Such a shared system can be made to be more scalable than our application-centric approach, but the low-level measurements it provides may not be as useful as the application-level measurements our approach uses. Further, the measurement granularity is determined by the system and may not be right for the application. In contrast, our approach can concentrate its measurements where they are most appropriate. Another point is that communication resources which could otherwise be put to use by the application will instead be used to maintain the consensus. Finally, to keep the amount of communication manageable, measurements will take longer to propagate to all hosts as the number of hosts grows.

It is important to note that mixed approaches are possible. For example, a shared measurement system could provide input to our system for hosts that we have not run on yet. Alternatively, one could build a system that shared application-level information. Finally, a partial reservation scheme (supporting, say, reservations of network bandwidth) could also be used in conjunction with our system.

7 Preliminary results: load trace analysis

This section discusses an extensive analysis of a large number of fine grain host load traces which show a number of properties that strongly suggest that history-based prediction is feasible for predicting load. These properties include self-similarity, epochal behavior of local frequency content, and relatively simple phase space behavior. Self-similarity implies a long-memory process where the future depends (in part) on a long stretch of the past in a possibly complex way. This dependence provides hope that history-based prediction can work, but instills the fear that the relationship between the past and the future may be too complex to glean. However, the epochal behavior of the local frequency content of the load signal suggests that the relationship may be found tractably. By epochal behavior of the local frequency content, we mean that the manner in which the load varies (the frequency content of the load signal) is relatively stable for long periods of time (epochs.) Because each epoch is long, we have an extended opportunity to understand the stable behavior of the load within it. That the phase space behavior of load is relatively simple further suggests that it can be predicted. In particular, load appears to spend much of its time in a small number of well differentiated states, and the manner in which it switches between states seems to be qualitatively simple. Other researchers have found some of these properties in network measurements [47, 77, 32].

We begin by describing our measurement methodology. Next, we give an overview of our analysis, and then we discuss in depth the self-similarity, epochal behavior of local frequency content, and the phase space behavior of the load traces.

7.1 Measurement methodology

We developed a small tool to sample the Digital Unix one minute load average at one second intervals and log the time series to a data file. The 1 Hz sample rate was arrived at by subjecting DUX systems to varying loads and sampling at progressively higher rates to determine the rate at which DUX actually updated the value. DUX updates the value at a rate of $1/2$ Hz, thus we chose a 1 Hz sample rate. We ran this trace collection tool on 38 hosts in the CMCL and PSC environments for slightly more than one week in late August, 1997. All of the hosts were DEC Alpha machines and they form four groups:

- *PSC Hosts*: 13 hosts of the PSC's "Supercluster", including two front-end machines, four interactive machines, and eight batch machines scheduled by a DQS [44] variant.
- *Manchester testbed*: eight machines in an experimental cluster in the CMCL.
- *Compute servers*: two high performance large memory machines used by the CMCL group as compute servers for simulations and the like.
- *Desktops*: 15 desktop workstations owned by members of the CMCL.

7.2 Overview of analysis

The purpose of our analysis of these traces was to determine whether history based prediction was feasible for load. We began by computing gross statistical properties for each trace, including mean load, variance, minimum, and maximum. We do not report on these properties here due to space. However, one of these, mean load, is used, along with mean epoch length (Section 7.4), to partition the machines into useful groups in Figure 9. Next we examined the overall power spectra (periodograms) and autocorrelation functions of the traces. Each trace did show strong close range autocorrelation, which bodes well for prediction, and led us to the more fruitful analyses. In the remaining sections, we will concentrate on these, namely self-similarity, spectrogram analysis, and chaotic dynamics.

7.3 Self-similarity and Hurst parameter estimation

Intuitively, a self-similar signal is one that looks similar on different time scales. If a self-similar signal looks jagged over a 10 second time scale, it will also look jagged over a 1 second or 100 second time scale. Another way of visualizing this is to imagine smoothing the signal with an increasingly long window. While the smoothing will quickly blur small variations in the signal, the signal will strongly resist becoming uniform (constant) with increasing window size. Statistically, the variance in increasingly smoothed versions of a series resulting from sampling the signal will only slowly decline. Contrast this with a series derived from picking from a typical stationary probability distribution. The variance of such a series will decline quickly with smoothing.

Self-similarity is more than intuition — it is a well defined mathematical statement about the relationship of the autocorrelation functions of increasingly smoothed versions of certain kinds of long-memory stochastic processes. These stochastic processes model the sort of the mechanisms that give rise to self-similar signals. We shall avoid a mathematical treatment here, but interested readers may want to consult [47]

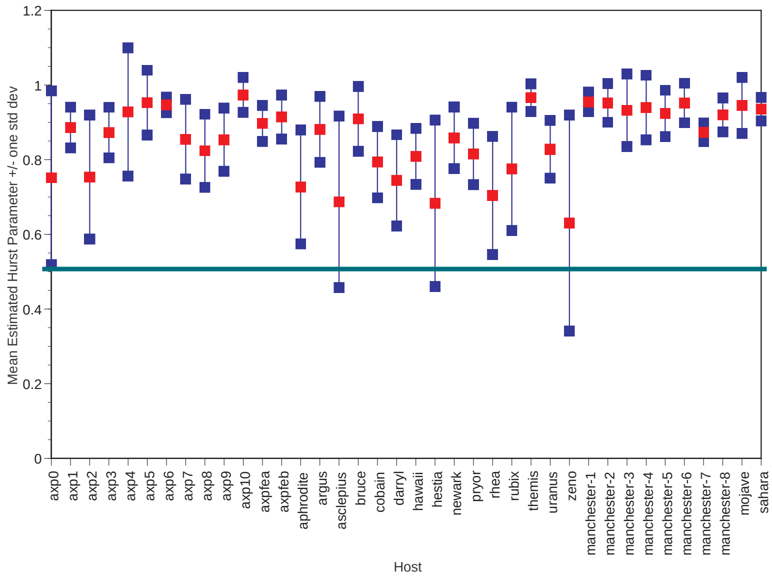


Figure 5: Mean Hurst parameter estimates for traces +/- one standard deviation.

or [55] for a treatment in the context of networking or [7] for its connection to fractal geometry. What is important is that self-similarity can arise from a long-memory process.

The idea of self-similarity and the statistic used to characterize it arose not in theory, but from the experimental physical sciences and engineering. Hurst [42] invented the notion and the statistic (now called the “Hurst parameter”) in order to characterize how the amount of water in reservoirs changes over time. He then applied it to summarize many other data series arising from natural processes. Many natural processes can be modelled in terms of the long-memory stochastic processes mentioned earlier, with the Hurst parameter describing the degree of dependence on the past. The Hurst parameter H ranges from zero to one. $H = 0.5$ indicates no self-similarity and is expected of a series generated from a stationary probability distribution where a sample is completely independent of all previous samples. $H > 0.5$ indicates self-similarity with positive near neighbor correlation, while $H < 0.5$ indicates self-similarity with negative near neighbor correlation. It is important to note that $H \neq 0.5$ does not imply that future depends entirely on the past, but rather that it has some dependence on the past.

We examined each of the 38 load traces for self-similarity and estimated each one’s Hurst parameter. Unfortunately, there is no consensus on how to best estimate the Hurst parameter of a series. The most common technique is to use several Hurst parameter estimators and try to find agreement among them. This is the approach we took in our analysis of the host load series. The four Hurst parameter estimators we used were R/S analysis, the variance-time method, dispersional analysis, and power spectral analysis. A description of these estimators as well as several others may be found in [7]. We implemented R/S analysis and the variance-time method using Matlab and performed dispersional analysis and power spectral analysis by hand on graphs prepared via Matlab. We validated each method by examining degenerate series with known H and series with specific H generated using the random midpoint displacement method. The dispersional analysis method was found to be rather weak for H values less than about 0.8 and the power spectral analysis method gave the most consistent results.

Figure 5 presents our estimates of the Hurst parameters of each of the 38 load traces. In the graph, each central point is the mean of each of the four estimators, while the outlying points are at +/- one standard deviation. Notice that for small H values, the standard deviation is high due to the inaccuracy of dispersional analysis.

The traces exhibit self-similarity with Hurst parameters ranging from 0.63 to 0.97, with a strong bias toward the top of that range. This is a promising result for prediction because it strongly suggests that future load values have a strong, albeit complex, dependence on past load values. This is not surprising for relatively quiescent machines, but it also makes sense for heavily loaded machines such as the PSC hosts, which spend long periods of time executing in particular program phases, then different programs, and on and on in to coarser and coarser timescales.

7.4 Epochal behavior of local frequency content

The observation we make in this section is that while the load changes in complex ways, the manner in which it changes remains relatively constant for relatively long periods of time. We refer to a period of time in which this stability holds true as an epoch. For example, the load signal could be a 0.25 Hz Sin wave for a minute and a 0.125 Hz sawtooth wave the next minute — each minute is an epoch. That these epochs are long is what is important. Long epochs are preferable over short epochs since there is more time in a long epoch to learn its behavior and use that knowledge to predict its future. The mean epoch lengths of our load series range from 150 to over 450 seconds, which means we may be able to take 1000s of samples of a particular epoch given the response times we require of our interactive applications (Section 2.)

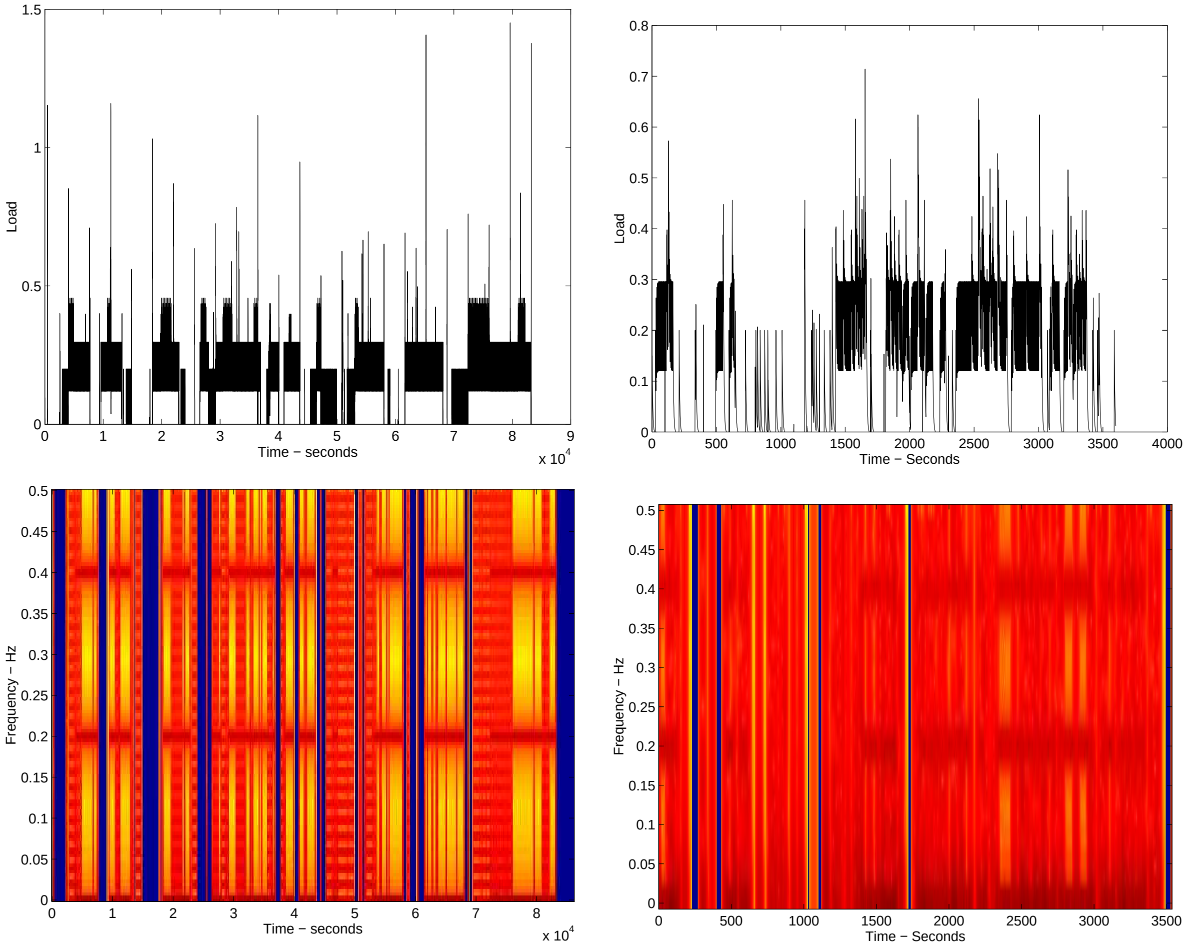
This observation ties nicely with the self-similarity results of the previous section. Those results support the feasibility of history-based prediction, but also raise concerns about its practicality. Could the long-memory process behavior implied by the result in fact be too complex to usefully predict? Could long past values affect current values so strongly that the history we would have to keep would be so long and the computation from it so expensive that prediction would become impractical? That the epochs we find are so long suggests that this is not the case and that history-based prediction is tractable.

A spectrogram representation of a load trace immediately highlights the epochal behavior we discuss in this section. A spectrogram combines the frequency domain and time domain representations of a time series. It shows how the frequency domain changes locally (for a small segment of the signal) over time. For our purposes, this local frequency domain information is the "manner in which [the load] changes" to which we referred earlier. To form a spectrogram, we slide a window of length w over the series, and at each offset k, we Fourier-transform the w elements in the window to give us w complex Fourier coefficients. Since our load series is real-valued, only the first $w/2$ of these coefficients are needed. We form a plot where the x coordinate is the offset k, the y coordinate is the coefficient number, $1, 2, \ldots, w/2$ and the z coordinate is the magnitude of the coefficient. To simplify presentation, we collapse to two dimensions by mapping the logarithm of the z coordinate (the magnitude of the coefficient) to color. The spectrogram is basically a midpoint in the tradeoff between purely time-domain or frequency-domain representations. Along the x axis we see the effects of time and along the y axis we see the effects of frequency.

Figure 6 shows the time domains (top) and spectrograms (bottom) of two series. The series on the left is a 24 hour trace on a PSC host, axp7, taken on August 19th and the series on the right is a one hour trace on a desktop, themis, taken from 2-3pm on August 20th. We examined the spectrograms of all the load traces both visually and analytically. These two are presented as representative cases. Also note that there is evidence of foldover from a too-low sampling rate in these signals. We are convinced that the problem lies in how often DUX actually updates its load value and not in our sampling. One explanation is that DUX does not update the load average whenever the length of the ready queue changes. Another is that it does not update it when running device drivers or other in-kernel tasks.

What is important to note about all of the spectrograms is what is very obvious about each of these examples - the relatively wide vertical bands. These indicate that the frequency domain of the underlying signal stays relatively the same for long periods of time. We refer to the width of a band as the duration of that frequency epoch.

That these epochs exist can be explained by programs executing different phases, programs being started



axp7, August 19, 1997 (24 hours)　　themis, August 20, 1997 2pm (one hour)

Figure 6: Time domain and spectrogram representations of load.

and shut down, and the like. The frequency content within an epoch contains energy at higher frequencies because of events that happen on smaller time-frames, such as user input, device driver execution, and daemon execution.

We can algorithmically find the edges of these epochs by computing the difference in adjacent spectra in the spectrogram and then looking for those offsets where the differences exceed a threshold. Having found the epochs, we can examine their statistics. The mean epoch length ranges from about 150 seconds to over 450 seconds, depending on which trace. The standard deviations are also relatively high (80 seconds to over 600 seconds.)

These long periods of relative stability are periods in which it is likely that a history-based prediction algorithm can collect enough information to characterize the stable frequency domain and make useful predictions. Also, because the epochs are long, we expect that after the prediction algorithm has converged, there will remain enough time to make use of its predictions before it needs to be retrained for the next epoch. Detecting the start of a new epoch quickly is an important problem. Querying a companion management information database or registry that keeps track of the coarse grain behavior of programs, such as when new programs enter the environment and when new program phases begin, may be one way to address it.

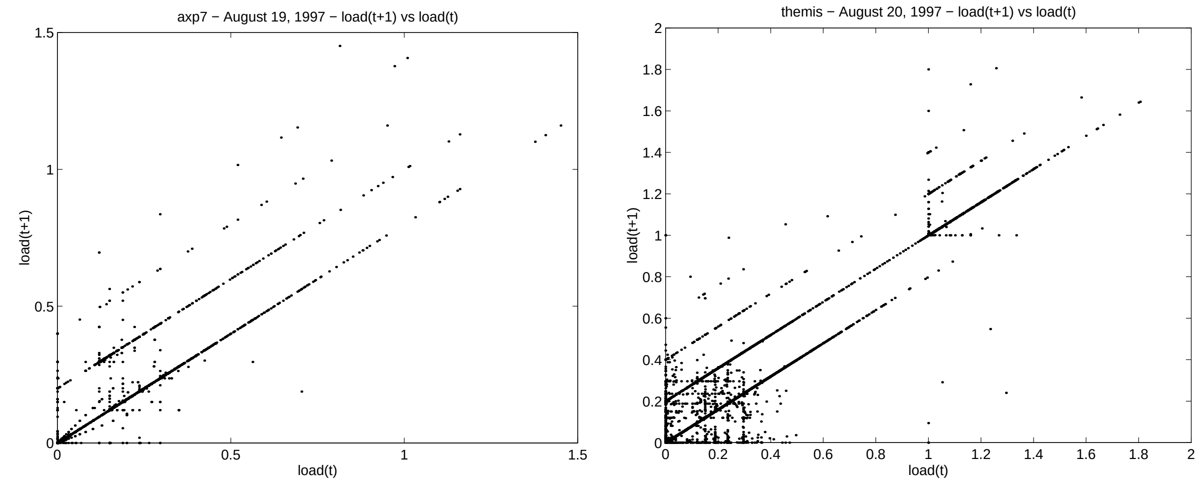


Figure 7: Phase space behavior.

7.5 Phase space behavior

An initial examination of the phase space behavior [3] of several of the load traces we have collected suggests that they arise from relatively simple chaotic systems. Plots of the systems' trajectories through their phase space suggest that the systems spend most of their time in a small number of states. Animations show that the patterns of transitions between states are often relatively simple. While the analysis presented here is far less detailed than for the previous sections, it nonetheless strongly supports that load can be predicted using history-based methods. Intuitively, the fewer the number of states the system has and the simpler its state transitions, the easier it is to predict.

In chaotic dynamics, a basic tool is a plot of the trajectory of the system through its phase space [54]. The dimensions of a phase space are the values in the system that change over time. If the system being examined is chaotic, meaning that small differences in the initial position in the phase space result in large differences in the trajectory, then the trajectory will tend to be quite complex and never exactly repeat itself, however it may be self-similar. A nonchaotic system will present a far simpler trajectory that repeats itself. Although the trajectories of chaotic systems are inherently more complex, they can often be characterized by "strange attractors" about which the trajectory orbits.

Trajectories are easier to see in a multidimensional space, but our datasets are unfortunately one dimensional. To form a two dimensional space, we plot the load time series against itself with a single unit delay - that is, we plot $l_2, l_3, \ldots$ versus $l_1, l_2, \ldots$. The series $\langle(l_i, l_{i+1})\rangle$ forms the trajectory through the space. This technique is known as the pseudo-phase space method [54]. Phase space plots for two load traces are given in Figure 7.

Another way of looking at these plots is as to how well the load at time t predicts the load at time $t+1$. For a perfect prediction, we would expect to see a single line of unity slope and zero intercept. For a linear relationship, we would expect to see a single line of some other slope and intercept. Instead, we see several lines and a number of scattered points. The vast majority of the data is in the lines. Clearly, if we knew each line's slope and intercept, which line we were on, and that we would remain on it during the measurement of the next load value, we could make a perfect prediction of the next load value from the current one. Prediction would then involve finding the lines, which does not appear to be especially difficult, and determining when the system moves from one line to another.

[3] Here we adopt the terminology of chaotic dynamics, where phase is the continuous equivalent to state. This should not be confused with the phase component of the Fourier transform of a signal.

The lines represent the few states the system spends the majority of its time in. The state transitions the system makes are quite clear when we animate its trajectory through the phase space. [4] The trajectory forms clear geometric figures, mostly triangles, whose vertices are on the lines we noted earlier. The state transitions and the movement of the geometric figures they aggregate to, appear eminently predictable. The centers of mass of these figures could be interpreted to be strange attractors. These features suggest strongly that history-based prediction is feasible for load.

8 Preliminary results: algorithms and evaluation for simplified problem

We developed history-based prediction algorithms to attack a simplified version of the dynamic mapping problem. In the simplified problem, we map only leaf nodes and ignore communication costs. Our measure of algorithm performance is the percentage of a large number of calls that execute within their bounds. The most significant result is that one of the algorithms we developed is both tractable and near-optimal under most conditions. This is strong evidence that the full problem can also be addressed using history-based prediction. In this section, we present extensive simulation results showing the performance of this and other algorithms in different instances of the simplified problem.

There are several other lessons to draw from our results: (1) Dynamic mapping can clearly make a significant difference in the percentage of mappings that meet their bounds; (2) Simple, statistical algorithms, even with randomization, are insufficient and behave inconsistently; (3) Complex algorithms such as neural networks work reasonably well but are clearly too expensive to use in practice. Our near optimal algorithm is simple enough to be practical, but is not so simple that it is useless.

Our approach to developing and evaluating algorithms was to build a trace-driven simulator that uses the load traces described in Section 7 to synthesize highly realistic execution times. Each algorithm predicts which host will most likely meet the required time bounds based on a past history of execution times. After the call is executed in simulation, the algorithm is informed of the actual execution time. The simulator also tries all possible mappings in order to determine how well an optimal mapping strategy can do. In addition, it computes the performance of a random mapping approach and of always mapping to the same individual host.

In the following, we first describe the simplified mapping problem in more detail. Then we describe the simulator and how execution time is determined from a load trace. This is followed by a description of the algorithms we implemented and tested. Finally, we present results for how the performance of the different algorithms depends on nominal execution time, relaxation of the upper time bound, and varying nominal execution times. We summarize the results for six different simulated computing environments that have different combinations of mean load and mean epoch as described in Section 7. We conclude by interpreting the results in light of the load trace analysis of the previous section.

8.1 Simplified mapping problem

We simplify the dynamic mapping problem posed in Section 5 by mapping only leaf nodes and ignoring communication costs and the costs of the running the prediction algorithm. Referring to the activation tree execution model of Section 4, we have $t_{map}, t_{incomm}, t_{outcomm}, t_{update} = 0$ and $t_{compute} = t_{computelocal}$. The thesis work will relax these simplifications and address the full dynamic mapping problem.

[4] To see an example animation, cd to /afs/cs/usr/pdinda/public and run demo.csh. You will need matlab.

8.2 Load trace-based Simulation

To develop and evaluate our algorithms, we built a simulator that allows us to simulate the execution of the loop

`for i=1 to N do MAP leaf_procedure() IN` $[t_{min}, t_{max}]$ `endfor`

where the nominal execution time, $t_{nominal}$, of `leaf_procedure` is either constant or chosen from an enumerated distribution. The symbols we use in this section are summarized in Figure 2. The nominal execution time is the time the procedure would take to execute under zero-load conditions on the slowest host. Since we are ignoring communication and mapping overheads, the procedure begins executing at t_{now}, the point in time the `MAP` call was executed. For the duration of the loop, the mapping algorithm is the same, so time advances according to the mapping decisions made by the algorithm under test, just as it would in the real world.

Given the load trace of the host the mapping algorithm has chosen, t_{now}, and $t_{nominal}$, we use a simple but very accurate model to determine the actual compute time, $t_{compute}$, of the node on that host at that time. First, consider a functional form of the load trace $\langle l_i \rangle$ so that $load(t) = l_i$ for $i\Delta \leq t < (i+1)\Delta$, where Δ is the sampling interval of the load trace. The actual execution time, $t_{compute}$ is the value which satisfies

$$\int_{t_{now}}^{t_{now}+t_{compute}} \frac{\beta}{1 + load(t)} dt = t_{nominal}$$

where β is the raw speedup of the host over the slowest host. We have found that the compute time estimates given by this model correlate strongly (> 0.99 coefficient of correlation) with the actual computation time.

For each simulated `MAP` , the simulator requests a host choice from the mapping algorithm based on the timing constraints, executes the call, and then informs the mapping algorithm of the actual execution time given its mapping.

Optimal mappings

Using the model for computing execution times from load traces, the simulator not only executes the procedure on the host the algorithm chooses, but on every host. This lets the simulator compute what an optimal mapping algorithm would do if invoked at this point: it picks a host whose execution time falls within the bounds, if one is available. It is important to note that simulated time evolves according to the algorithm under test, thus the measured performance of the optimal algorithm depends slightly on the algorithm under test. In practice, this varies only marginally across the algorithms we test. We present the performance of the optimal algorithm with respect to our near optimal algorithm, *RangeCounter(W)*. In this incarnation, the optimal algorithm is referred to as *Optimal(RC)*.

8.3 Prediction algorithms

We designed and implemented nine different history-based prediction algorithms for the simplified mapping problem. Most of the history-based algorithms are the simple statistical algorithms that one is lead to when initially thinking about the problem. A common failing of these is that they fall into modes where they refuse to “explore” the space of possible mappings and so can get “stuck” giving unfortunate mappings repeatedly. We added randomization to these initial algorithms to attempt to avoid this. The two remaining algorithms include a neural network algorithm and the *RangeCounter(W)* algorithm, which is nearly optimal most of the time. This section describes each of these algorithms. Figure 8 shows the asymptotic complexity and actual running times of all the algorithms. It also includes three other algorithms we use for comparison purposes, *Optimal(RC)*, which as described in Section 8.2, *Random*, and *Constant*. *Constant* simply always picks the same host while *Random* always picks a host at random.

	Complexity		Timingsμ-sec		
Name	Time	Space	(4,50,0.1)	(8,50,0.1)	(16,50,0.1)
Optimal(RC)	-	-	-	-	-
Constant	$O(1)$	$O(1)$	0.244	0.244	0.244
Random	$O(1)$	$O(1)$	0.498	0.537	0.546
Mean	$O(N)$	$O(N)$	0.957	1.532	2.710
WinMean(W)	$O(N)$	$O(NW)$	3.318	4.460	7.005
WinVar(W)	$O(NW)$	$O(NW)$	7.039	9.580	14.73
RandWinMean(W,P)	$O(NW)$	$O(NW)$	7.223	13.166	24.534
RandWinVar(W,P)	$O(NW)$	$O(NW)$	17.408	33.58	65.866
Confidence(W)	$O(NW)$	$O(NW)$	13.674	21.360	36.775
RandConfidence(W,P)	$O(NW)$	$O(NW)$	26.951	52.869	102.423
RangeCounter(W)	$O(N) + O(W)$	$O(NW)$	8.500	9.551	11.960
NeuralNet(W)	$O(N^2W^2)$	$O(N^2W^2)$	41807.630	175543.700	773379.700

Figure 8: Algorithm complexity and timings. The numbers (N, W, P) refer to the number of hosts, the window size, and the randomization probability. Timings were collected on a DEC 3000/400.

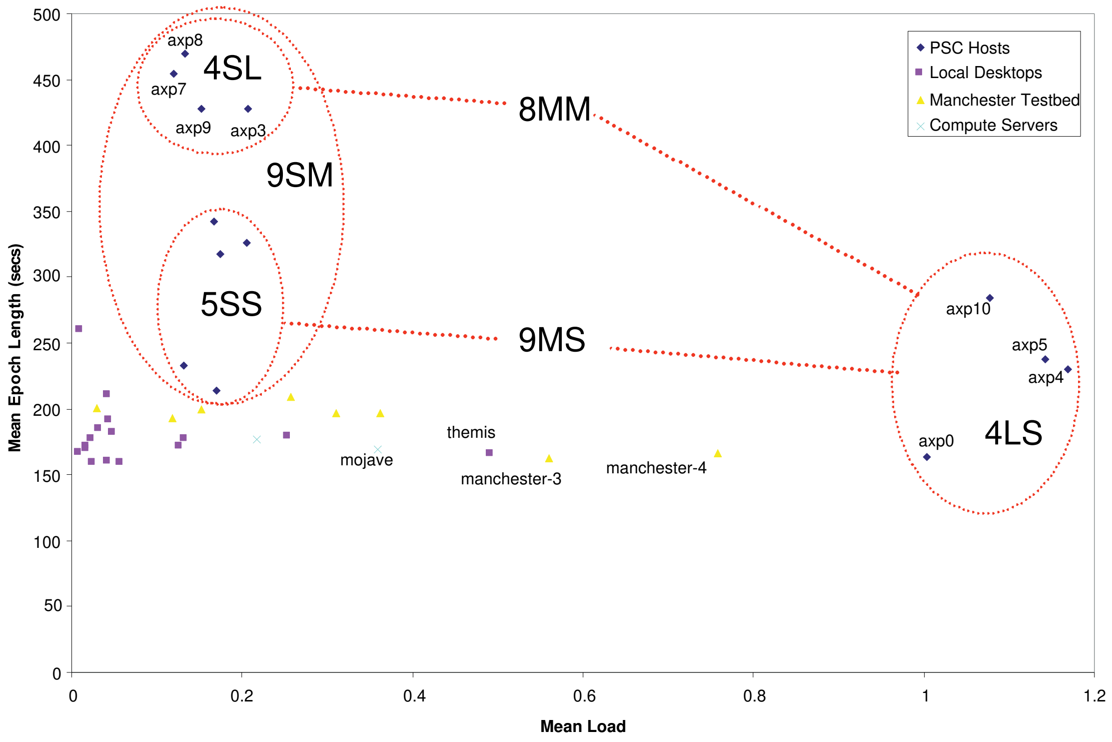
The simplest algorithms include *Mean*, *WinMean(W)*, and *WinVar(W)*. For each host *Mean* computes the mean execution time over the entire history of executions on that host and chooses the host with the lowest mean. *WinMean(W)* is similar, except the mean is computed over the last W executions on the host. *WinVar(W)* chooses the host with the lowest variance in execution time over the last W executions. *RandWinMean(W,P)* and *RandWinVar(W,P)* select a random host with probability P and act like their non-randomized counterparts with probability $1 - P$.

Confidence(W) assumes that execution times are picked from a normal distribution.[5] It computes the sample mean and variance over the last W executions on the host and parameterizes a normal distribution with these values. The host whose normal has the highest cumulative probability over the $[t_{min}, t_{max}]$ range is chosen. The *RandConfidence(W,P)* algorithm chooses a random host with probability P and performs like *Confidence(W)* with probability $1 - P$.

The *RangeCounter(W)* algorithm can be thought of as a non-linearly filtered version of the *Confidence(W)* algorithm. Each host is assigned a quality value, initially zero. To choose a host, the following steps are taken: If no host has a quality value significantly higher than zero, a random host is chosen, otherwise the host with the highest quality value is chosen. Next, the quality values of all hosts are reduced (aged) by a small amount. If the mapping is successful, then the chosen host's quality value is increased by the inverse of our statistical confidence in it. As with the *Confidence(W)* algorithm, we compute the sample mean and variance of the past W executions, parameterize a normal distribution with them, and compute the cumulative probability of that distribution over $[t_{min}, t_{max}]$. We increase the host's quality value by the *inverse* of the cumulative probability. If the mapping is unsuccessful, we divide the chosen host's quality value by two. *RangeCounter(W)* rewards good choices based on how unlikely they appear to be able to bear fruit and punishes bad choices equally. The idea is to encourage exploration and risk taking, but to react quickly and decisively to failure. The aging process discourages complacency.

The *NeuralNet(W)* algorithm consists of a multilayer feed-forward neural network [53] where each host contributes W inputs, the durations of the last W executions on that host. For N hosts, the intermediate layer has $NW/2$ nodes and the output layer has N nodes. The network is trained to drive the output representing the best host to the highest output value. To choose a host, the inputs are propagated forward through the network and the host with the highest corresponding output value is chosen. If the choice is successful, we reinforce the chosen output to unity and the remaining outputs to their present values and propagate these

[5] A variety of other distributions were also examined.



	Load Category		
Epoch Category	Small	Mixed	Large
Short	5SS	9MS	4LS
Mixed	9SM	8MM	
Long	4SL		

Figure 9: Test configurations for PSC hosts

outputs back through the network. Similarly, if the choice is unsuccessful, we reinforce the chosen output to zero and the remaining outputs to their present values and propagate back. Essentially, Each choice/response pair is an iteration of the BACKPROP neural network training algorithm. We continuously train the network in operation.

8.4 Simulated environments

We constructed six different configurations of hosts from the load traces discussed in Section 7. The configurations consist of traces of hosts with different mean loads and different mean epoch lengths. In Figure 9 we plot each load trace according to its mean load and its mean epoch length. The six configurations we discuss here are then highlighted, labeled, and tabulated. We categorize the load of a configuration by whether it contains only machines with low (≈ 0.2) mean loads, only machines with high mean loads (≈ 1.1), or a mixed combination of machines in the former categories. Similarly, we categorize the epoch length of a configuration by whether all of its machines have short (150-300 seconds) epochs, long (400-500 seconds) epochs, or a combination of the former. The labels consist of a number and two letters. The number represents the number of hosts in the configuration, the first letter represents whether the load category is (S)mall, (M)ixed, or (L)arge, and the second letter indicates whether the epoch category is (S)hort, (M)ixed, or (L)ong. Notice that there are nine combinations, but our data limits us to exploring only six of the combinations.

The configurations presented here contain only the PSC hosts. We have also done limited simulations using other configurations involving the remaining machines, but there is no space to present them here. The advantage of using the PSC hosts is that they nicely cover most of the space formed by the load and epoch categories.

For each of the six configurations, we looked at eight different values of the nominal execution time $t_{nominal}$, and twelve different values of the upper timing bound t_{max}, ranging for each case from $t_{nominal}$ to three times $t_{nominal}$. t_{min} was set at zero. This forms 576 different cases and we simulated 100,000 consecutive calls for each case, for a total of 57.6 million simulated calls. In addition, for each configuration, we simulated what happens when $t_{nominal}$ is picked randomly from a set of four values (25,50,75,100 ms) on each iteration, and t_{max} is varied as before. This forms another 72 cases and 7.2 million calls.

8.5 Algorithm performance

In this section, we present three slices through our simulation results to make presentation tractable. The first two slices show how the algorithms' performance varies with nominal execution time and the upper time bound. The third slice shows how the performance varies with the upper time bound when the nominal execution time is chosen randomly from four different values. The performance of an algorithm is the percentage of calls that meet the time bounds.

Our discussion covers all of the simulations. However, in order to simplify our presentation, we show graphs for only two representative configurations, 8MM and 4LS. As can be seen in Figure 9, 8MM contains eight machines with mixed load and epoch lengths, while 4LS contains four machines with high loads and short epoch lengths. To show the effect of nominal execution time, only 8MM is shown since 4LS does not provide any additional insight. The other configurations show similar results.

For the algorithms that have parameters, namely W, the window of previous execution times recorded for a host, and P, the probability of a random mapping (for the randomized algorithms), we have fixed the parameters to $W = 50$ (10 for *NeuralNet(W)*) and $P = 0.1$. The performance of the algorithms given these values is representative of how they perform with the other, non-extreme, values we studied. The relative performance of the algorithms is not very sensitive to the choice of these values.

Effect of nominal execution time

Figure 10 shows how the different algorithms perform when mapping nodes with different nominal execution times in a representative configuration, 8MM. The other configurations yield similar results. The nominal times range from 50 milliseconds to 60 seconds. The x-axis is the nominal time, $t_{nominal}$, ranging from 50 milliseconds to 60 seconds, and the y-axis is the percentage of calls that meet the bounds. The bounds are set to $[0, t_{nominal}]$.

Notice that there is a substantial gap between the performance of the optimal mapping algorithm (*Optimal(RC)*) and the performance of random mappings (*Random*). Although we have elided the results for clarity, it is also never the case that always choosing the same host works well. We find similar results for the other five configurations, with differences ranging from 40 to 80 percent. This suggests that a good mapping algorithm can make a large difference.

Interestingly, the simple statistics-based mapping algorithms and their randomized variants perform rather badly for the most part, some worse than even a random mapping. Further, they tend to behave inconsistently. For example, *WinVar(W)* is about 35% better than *Random* at 50 milliseconds, then abruptly declines with increasing $t_{nominal}$ only to improve later and finally become completely ineffectual at 60 seconds. In another configuration, we find it near optimal for some nominal times and yet collapsing to worse than *Random* for nearby times. On the other hand,*NeuralNet(W)* and *RangeCounter(W)* behave consistently well and

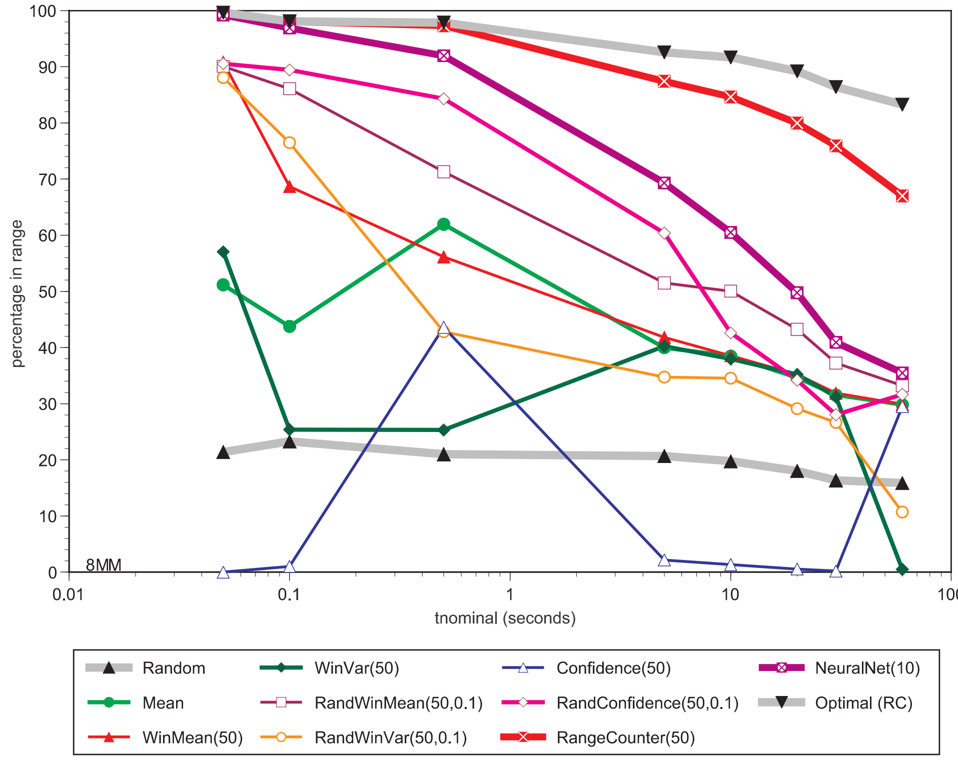


Figure 10: The effect of varying nominal execution time for the 8MM case. The bounds are $[0, t_{nominal}]$.

near-optimally for short nominal times less than one second. In fact, *RangeCounter(W)* is the best algorithm for all configurations and nominal times except for the 60 second time in the 4LS configuration.

Effect of bounds

The effect of loosening the upper bounds on the performance of the algorithms is quite interesting. In Figure 11 we show the effect in the 8MM configuration and in Figure 12 we show the effect in the 4LS configuration. The x-axis in each graph is the upper time bound and the y-axis is the percentage of calls that meet the bound. The nominal execution time is fixed at 100 milliseconds and the upper bound varies from 100 milliseconds to 300 milliseconds. The remaining four configurations show effects similar to the 8MM configuration (Figure 11). If we scale the nominal time and bounds, we see similar graphs in each of the configurations. For example, the graph for the 8MM configuration with a 10 second nominal time whose x-axis ranges from 10 seconds to 30 seconds looks similar to Figure 11. The choice of a 100 ms time for this set of graphs was no accident. 100 ms marks the transition between imperceptible delay and perceptible delay. To be able to place a tight bound on a 100 ms task is important because even small extensions in the delay have large consequences.

In Figure 11, notice that there is great differentiation between the optimal algorithm and a random mapping as the upper bound gets tighter. The same holds true in Figure 12 for bounds greater than 200 ms. For this configuration, there are so few machines that can meet the tighter bounds that the gap between *Optimal(RC)* and *Random* declines severely with bounds tighter than 200 ms.

In both figures, the simpler algorithms require a reasonably large amount of additional slack in the upper bound before their performance catches up to *RangeCounter(W)* and *NeuralNet(W)*. This is also the case for the other configurations, although the difference tends to be slightly less for configurations with only unloaded machines. In Figure 11, the simpler algorithms require between 25 and 100 milliseconds of extra

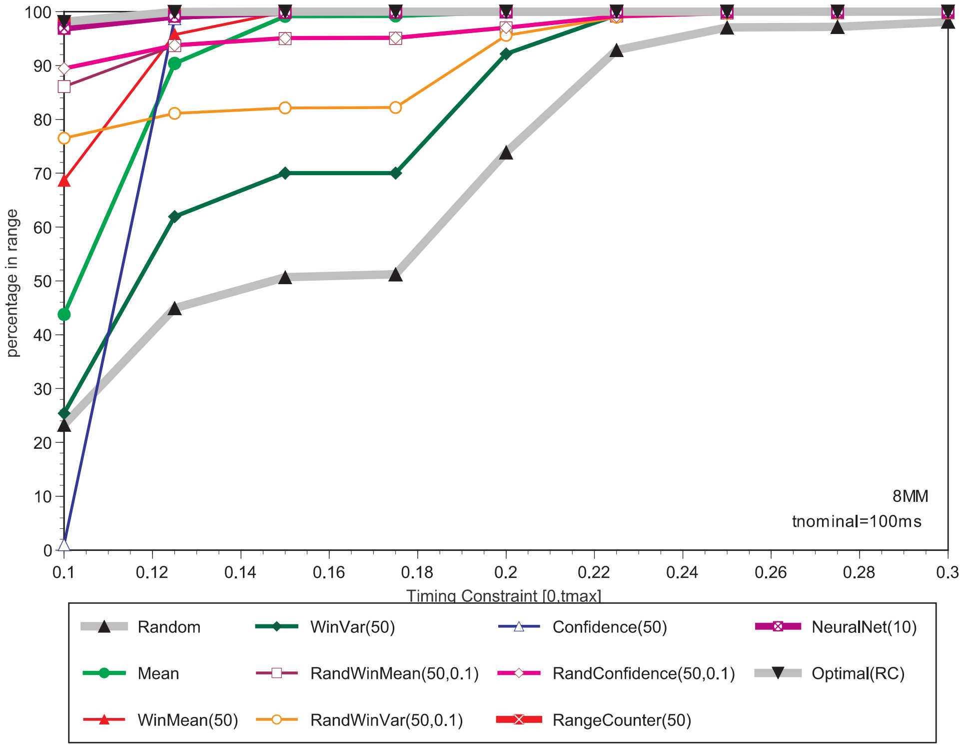


Figure 11: The effect of different timing constraints for the 8MM case. $t_{nominal} = 100ms$ and the constraint is $[0, t_{max}]$ where t_{max} varies from 100ms to 300ms.

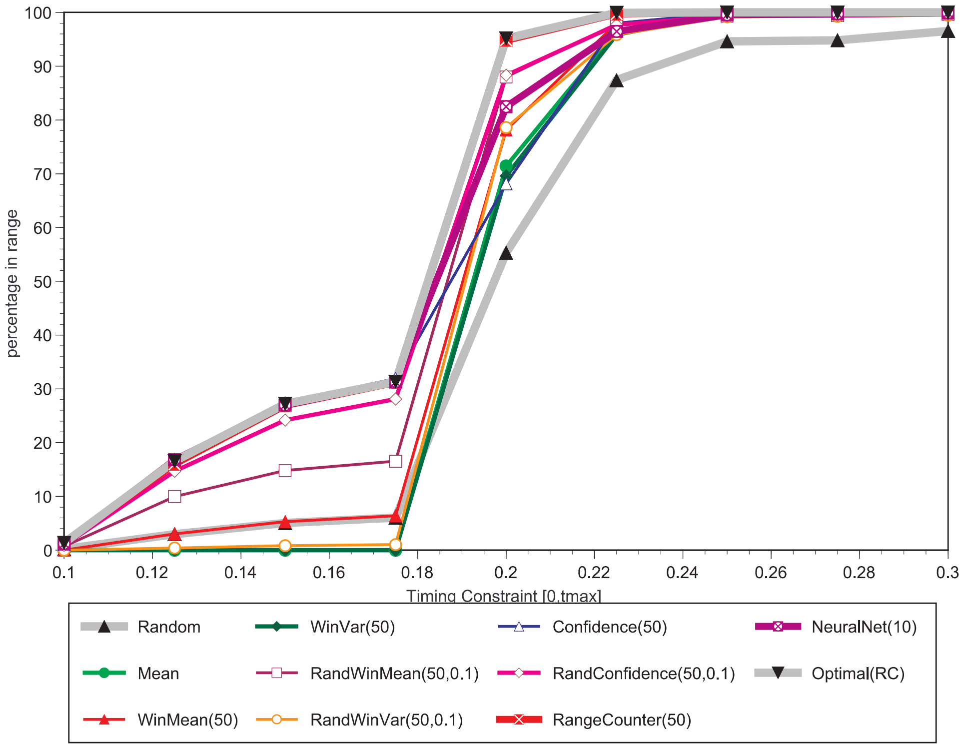


Figure 12: The effect of different timing constraints for the 4LS case. $t_{nominal} = 100ms$ and the constraint is $[0, t_{max}]$ where t_{max} varies from 100ms to 300ms.

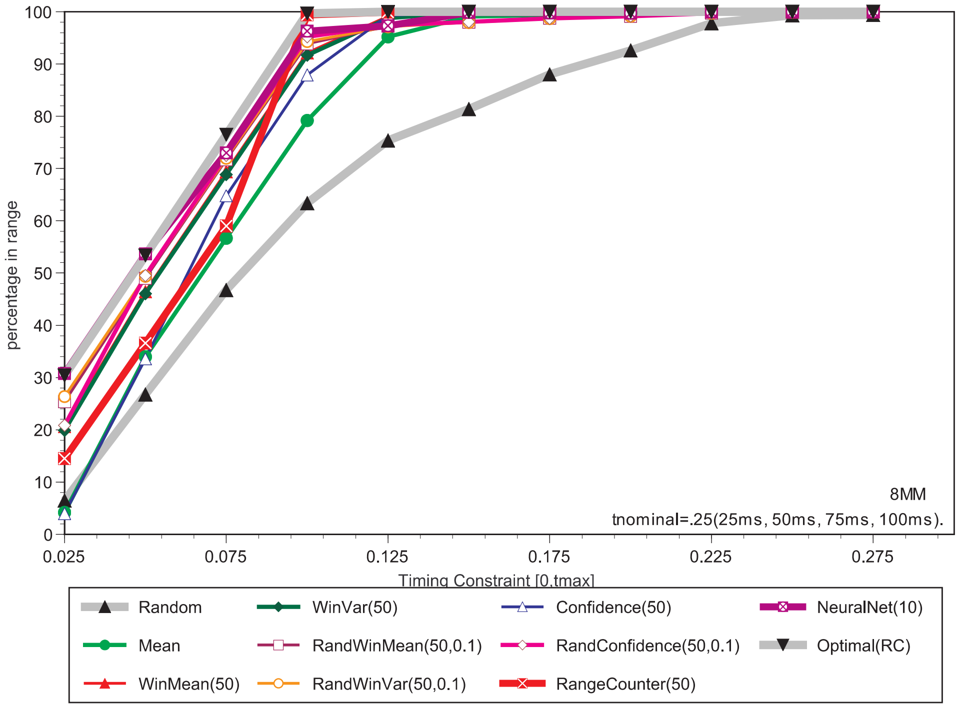


Figure 13: The effect of different timing constraints for the 8MM case. $t_{nominal} = 100ms$ and the constraint is $[0, t_{max}]$ where t_{max} varies from 100ms to 300ms.

slack to get to within an absolute 10% of the performance of *RangeCounter(W)* and *NeuralNet(W)*. This is a significant amount of time. For Figure 12, we see a great upsurge in performance as the bound is relaxed to about double the nominal execution time. Before this point, there are very few opportunities in which it is possible to meet the bound. However, *RangeCounter(W)* manages to ferret out these opportunities and manages to closely approximate the optimal algorithm's performance across the board. It behaves well even when choices are very constrained.

Effect of varying nominal execution time

Figures 13 (8MM configuration) and 14 (4LS configuration) show the effect of loosening the upper bound when the nominal execution time is picked randomly from a set of values. In each graph, the nominal time is randomly picked from 25, 50, 75, and 100 milliseconds for each call. Intuitively, this can be seen as following different control paths through the procedure. The x-axis is the upper time bound and the y-axis is the performance. We sweep the upper bound from 25 ms to 300 ms. The linear upward trend on the first part of each graph is due to the varying nominal time. At 50 milliseconds, only about 50% of the calls have a nominal time of 50 milliseconds or less, so even the optimal algorithm can only manage to map 50% of the calls successfully, even given totally idle machines.

Notice the separation between the optimal algorithm and random mapping, suggesting opportunity for a mapping algorithm. The separation is smaller than in the previous cases, since at any particular time bound except for 25 ms, there are some number of calls that have significant slack. For example, at 50 ms, 25% of the calls have a nominal time of 25 ms and thus have a 2x slack. This also benefits the simple algorithms and decreases the spread in performance. However, the spread can still be significant. For example, in the 8MM configuration (Figure 13), the spread is as high as an absolute 40% of the calls.

An important point here is that *RangeCounter(W)*, although it does well, behaves somewhat erratically in

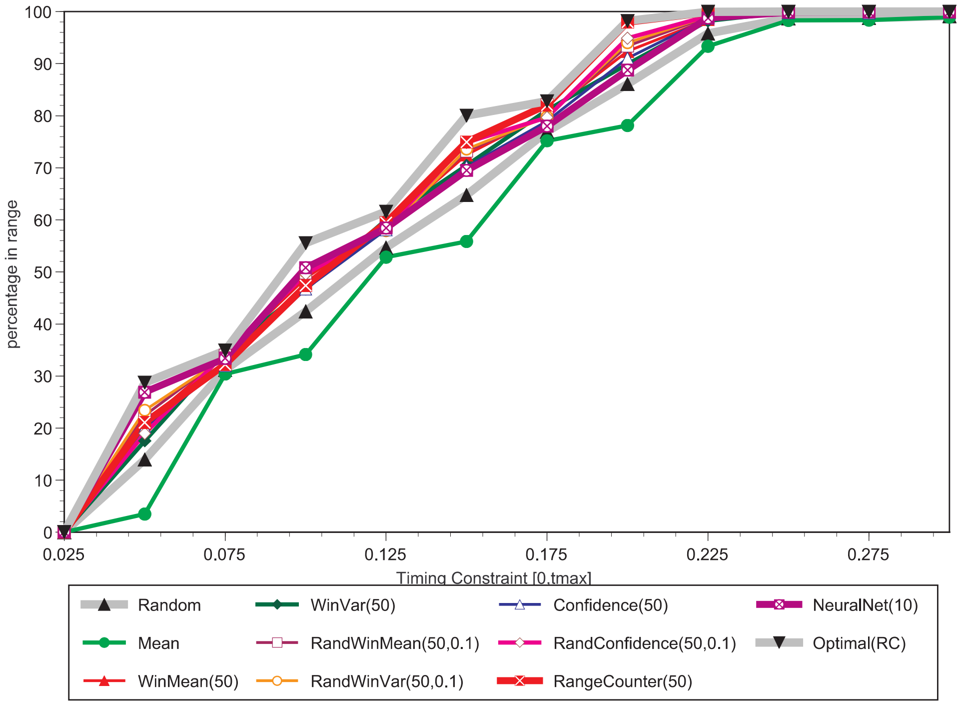


Figure 14: The effect of different timing constraints for the 4LS case. $t_{nominal} = 100ms$ and the constraint is $[0, t_{max}]$ where t_{max} varies from 100ms to 300ms.

the face of the varying nominal times and is not clearly superior to the other algorithms until the bound has been relaxed to encompass the longest of the different nominal times. That is, when some varying nominal times exceed the bound, *RangeCounter(W)* can behave badly. In some cases, it is as much as 20% down from the optimal algorithm and *NeuralNet(W)*. This presents opportunity for improvement. Indeed, this behavior will become more important when we begin to map activation trees, since we cannot expect all the nodes to require the same nominal time to execute. Far from being sorrowful, this is actually an opportunity for the thesis research.

Conclusions

There are several conclusions to be drawn from our results. The first is that a mapping algorithm can make a large difference. There is a large spread between the performance of an optimal mapping algorithm and mapping randomly or always mapping to the same host. The difference can be as high as 80%. The second conclusion is that simple statistics-based algorithms are insufficient. In some cases, simple algorithms such as *Mean*, *WinMean(W)*, *WinVar(W)*, and *Confidence(W)* can actually perform worse than a random mapping. Worse, they can behave erratically in the face of different nominal execution times. Adding randomization to these algorithms smoothes them and improves their overall performance, but it still does not make them competitive with the best algorithms, *NeuralNet(W)* and *RangeCounter(W)*.

The load trace results we presented in Section 7 can shed some light on why the simple statistical algorithms perform so badly. Recall that the load traces exhibit self-similarity (Section 7.3) and that their phase space behavior is relatively simple and ordered (Section 7.5.) Clearly, consecutive load values are *not* chosen independently according to some stationary probability distribution function. However, this is precisely the underlying assumption made by the simple statistical algorithms! Further, even if, in the long term, some pdf does emerge, it is highly unlikely that this pdf is applicable over the short term, given the epochal behavior discussed in Section 7.4. Of course, we might expect that with a sufficiently long nominal time, the simple statistical algorithms might work better, but even there there is some doubt. Unfortunately, the histograms of the load traces (which we elide for space reasons) do not appear to be fit any analytic distribution we are aware of.

NeuralNet(W) and *RangeCounter(W)* appear to be better able to deal with the properties exhibited by the load traces because they behave non-linearly. This is probably a great benefit when transitioning from one epoch to the next because it suggests that the internal state of the algorithms will change drastically at these points, which is, of course, what we desire. Another interesting point is that as the nominal execution time grows, there are fewer and fewer training examples per epoch. This offers a possible explanation for why *NeuralNet(W)*, which behaves near optimally for short nominal times, quickly declines as the nominal times grow. Often, as neural networks are trained, they tend to converge only slowly to a good set of weights. After some large number of training sets, there is an abrupt convergence, beyond which any additional training tends to make the neural network slowly worse. We suspect that what we are seeing is that with a long enough nominal time there is simply an insufficient number of training samples in an epoch for *NeuralNet(W)* to make that abrupt convergence.

The most important conclusion is that *RangeCounter(W)* exhibits the best performance in nearly every case, and behaves relatively smoothly. For nominal times less than one second, it behaves almost optimally, while its performance degrades gradually for longer nominal times. Because it is near optimal even with the tightest bound, it is no surprise that it remains so as we relax the bound. Further, while *NeuralNet(W)* is also near optimal under conditions of tight bounds and short nominal times, none of the other tractable algorithms are especially close, and some require considerable loosening of the upper bound before they become anywhere near optimal. The failings of *RangeCounter(W)* are that behaves suboptimally in the presence of varying nominal times and very long nominal times. In some sense, *RangeCounter(W)* tends to become specialized too quickly, which is the reverse of *NeuralNet(W)*'s problem. *NeuralNet(W)* seems to remain sufficiently general to do well with varying nominal times. These imperfections present opportunity for future research.

9 Outline of thesis

In my thesis, I will extend the work I have done on the simplified dynamic mapping problem to attack the full dynamic mapping problem. There will be six stages to this work. The first stage will involve collecting extensive activation tree and network traces for use in trace-driven simulation. In the second stage, I will analyze and characterize these traces, looking for properties that may be useful for history-based prediction. The third stage will involve extending the trace-driven simulator I developed for my earlier work to use the traces. At this point, I will have a simulation environment in which host computation times, communication times, and activation tree structure will have high fidelity to the real world. In the fourth stage, I will use the simulator to develop a history-based dynamic mapping algorithm to solve the full problem. In the fifth stage, I will evaluate the algorithm in simulation and I will incorporate it into the LDOS system to show it works in a real system. In the final stage, I will write the dissertation document. The remainder of this section describes the stages in more detail. A timeline for these stages is given in Figure 9 and the remainder of this section describes the stages in more detail.

Trace collection

In order to simulate realistic communication times and activation tree structures, I will first collect extensive traces of real networks and real applications. These traces will complement the host load traces I have already collected and allow me to simulate realistic environments.

Activity	Artifacts	Duration
Trace collection	measurement tools, traces	3 Months
Trace characterization	models, trace families	3 Months
Simulator extension	simulator for full problem	2 Months
Algorithm development	Mapping algorithm	5 Months
Evaluation	Benchmarking results, LDOS implementation	2 Months
Writing/Defense	Defended thesis	3 Months
Total Time:		18 Months

Figure 15: Thesis timeline

I have already devised a methodology for collecting traces of activation trees in Microsoft Windows applications. I have written a tool based on the DyninstAPI run-time dynamic instrumentation library [40, 39] that lets me instrument the entry point and exit points of any procedure in a Windows COFF binary. An input file lets me specify which procedures to instrument. By adding calls to a dynamically linked logging library at entry and exit points, I can log activation trees to a file. The nodes of these trees are annotated with the time they started executing and the duration of execution. In addition to capturing the tree structures I require, by running these instrumented applications on a host with controlled (zero) load, the durations stamped on each node will reflect the nominal execution times that the simulator needs to determine actual running time given different load conditions.

In addition to the activation trees, it is also necessary to capture what data each procedure call references, since it will be necessary to account for communicating this data when simulating running these calls remotely. One way of collecting this information is to limit ourselves to codes whose interfaces are specified via an interface definition language or an interface repository such as in CORBA or DCOM. A more practical approach is probably to use virtual memory mechanisms to determine which pages each procedure references. I am currently looking at the user-level guard page mechanism in Windows NT for this purpose. Intuitively, on entry to a procedure, we can mark all the process's pages with the guard bit. Then each page the procedure references will cause an exception that we can catch. The set of referenced pages is a conservative estimate of the data the procedure references. Another approach to collecting this data is to instrument every load and store in the program. Whichever approach is used, it will be important to map the logged data reference behavior of a procedure call to the actual data movements that the remote execution facility would perform in order to execute the call remotely. For example, a trace of guard page faults corresponds directly to the data movements a simple page-based DSM system would have to undertake. However, an RPC-based system could move less data (because not all data on a page may be referenced) or more data (because data for *all* possible execution paths must be moved) than the trace suggests.

Extensive instrumentation will slow down the program considerably, which is a concern because it is interactive, and the user may behave differently if it runs slowly. However, it is important to note that we can collect our measurements in a two stage process. In the first stage, the user would execute an uninstrumented program with a monitoring tool which would only collect the messages being sent to the program. This would only marginally affect the performance of the program. In the second stage, the log of messages would be fed to the instrumented program, which could run as slowly as necessary to collect the activation trees and data reference behavior.

I am still undecided as to how to collect network traces. The purpose of these traces is to enable the simulator to answer the question "if I move N bytes of data from host i to host j at time t, how long will it take?" One possibility is to collect packet traces (records of the form (time,sender,receiver,size)) on a broadcast Ethernet network using a packet sniffer. Durations of additional traffic can then be computed either at the packet level or by the residual bandwidth. I have experience using this methodology in prior work characterizing network traffic of Fx programs [19]. Another possibility is to use the Remos system [17] which is currently being developed by the Remulac group or by in-switch monitoring using the active networking facilities being developed by the Darwin group [73]. While these approaches would allow me to collect data on more sophisticated ATM and switched Ethernet networks, it is unclear when they will be available and in what form. Another issue is the granularity at which measurements will be able to be done in these systems. Since I am concerned with relatively fine-grain communication, a measurement system that would provide only coarse information would be non-ideal. It may also be possible to use existing network trace databases or to generate a trace from an existing model.

The artifacts produced at this stage will consist of an infrastructure for collecting host, network, and activation tree traces, as well as a large collection of traces.

Trace characterization

In the next stage of my thesis work, I will characterize the network and activation tree traces I have collected looking for properties potentially beneficial to history-based prediction.

The way I analyze the network traces will mirror the approach I have taken in examining host load traces in my current work. In particular, I will use the tools of statistics, time series analysis, spectral analysis, self-similarity, and chaotic dynamics. I expect to be able to leverage the fairly extensive work that has been previously done by myself and others on characterization of aggregate traffic.

Activation tree traces are more complex than a simple time series and I am not sure exactly how to approach them at this point. In terms of solving the dynamic mapping problem, what is important is to know how to divide the bounds placed on a parent node among its children. A useful characterization will facilitate predicting how many children a node has and how large the subtree rooted at each of those children is.

The artifacts produced at this stage will be parameterizable models that can be fit to trace data, and classification of the traces into families based on the parameters. Members from the families can then be selected to form benchmarks.

Simulator extension

Now having a useful family of load, network, and activation tree traces, I will extend the simulator I have built to support them. I will develop a simple method to accurately compute communication time given the size of the message and a trace of the traffic on the network. The part of the simulator that produces nodes to be mapped will be replaced with one that can use the activation tree traces. I may also rewrite other portions of the simulator to make it more modular and adaptable to other related simulation studies - adding utility function support, for example. The result of this work will be a simulator well suited to attacking the full dynamic mapping algorithm.

Algorithm development

The heart of the thesis will be in developing an algorithm to solve the dynamic mapping problem I posed in section 5. In my current work, I have developed a good algorithm for a simplified version of the problem where communication is free and only leaf nodes are to be mapped. I hope to complement this algorithm with an algorithm that uses history-based prediction to divide the bounds placed on a parent node among its children. These predicted bounds will represent the unexecuted part of the activation tree. I am not firmly tied to the algorithm I have developed and it may be necessary or desirable to take a different approach. The figure of merit for a mapping algorithm will be the percentage of bounds it meets for a family of activation tree traces on several different simulated environments. A good algorithm will score close to the optimal as defined by simulating each possible mapping for sufficiently small activation trees, and each possible node choice given prior choices made by the algorithm under test for larger trees.

The questions I want to answer include

- How can we predict host and network properties, execution times, and whether bounds will be meet?
- What history is most fruitful to collect, host and network properties, execution time, or whether bounds are meet?
- Is predicting activation tree structure useful and how can we do it?
- Can we exploit the activation tree's structure by being more exploratory early in the tree?
- How deep in the tree should we go before we stop mapping and leave the subtree to execute entirely on the current host?
- Which single-node mapping algorithm should be used for a particular node?

Although they are interesting issues, I don't intend to look at caching strategies to make expensive algorithms feasible or how to incorporate input from external monitoring tools. I also will not look at how a reservation system changes matters. Another issue is procedure calls, such as calls to a windowing API, which must execute on one particular host. I will assume that all calls can execute on any host.

Evaluation

Since my simulation approach is so close to a real environment, I intend to do as much of the evaluation as possible using the simulator. I will develop benchmarks from the families of activation tree, network and host traces I will have collected and then test the algorithm's performance on these benchmarks. I hope to be able to draw connections between benchmark properties and the performance of my algorithm, at least to the extent that I can provide insight as to how to set any parameters the algorithm may have. The hope is to generalize the algorithm to the greatest extent possible.

I intend to compare this algorithm with other approaches. I believe that comparing to an approach based on a shared measurement system and simple heuristics (eg, choose the least loaded host) should be feasible. Certainly it will be possible to evaluate the effect of sharing information between hosts.

In addition, I will incorporate the algorithm into my LDOS distributed object system to show that my approach can work in a real system. I may also do some limited evaluation using this system, but at this point I am not sure what form this will take.

Writing and defense

In the final stage, I will write up my results and defend them.

10 Thesis contributions

The contributions of this thesis will include algorithms, models, evaluation, and artifacts. The contributions of my work to date include the following:

- **LDOS:** I developed a lightweight distributed object system to better understand modern remote execution mechanisms and procedure execution times on remote systems.

- **Load traces:** I built tools to collect host load traces and collected over a week of fine-grain host load data on a wide variety of machines (Section 7.1.) As far as I am aware, there is no such family of traces available to the research community.

- **Load trace analysis:** I performed an in-depth analysis of the collected load data, presented in Section 7. As far as I am aware, this is the first analysis to identify the following important properties of host load:

  - **Self-similarity** (Section 7.3)
  - **Epochal behavior of local frequency content** (Section 7.4)
  - **Simple phase space behavior** (Section 7.5)

- **Compute time model:** I created and validated a model for determining compute time from a load trace (Section 8.2)

- **Simulation infrastructure:** Using the compute time model, I developed a trace-based simulator for a simplified version of the dynamic mapping problem which ignores communication costs and maps only leaf nodes (Section 8.2.)

- **Algorithms:** I developed nine different mapping algorithms for the simplified mapping problem (Section 8.3.)

- **Evaluation:** Using the simulator and six representative groups of hosts, I vigorously evaluated the mapping algorithms for the simplified dynamic mapping problem (Section 8.5) and found the following results:

  - **Mapping algorithms matter.** Mapping calls to randomly selected hosts or always to the same host results in significantly fewer of the calls executing within their time bounds than is optimal.
  - **Simple statistical algorithms are insufficient.** The performance of such algorithms is suboptimal and highly dependent on the tightness of the time bounds, the nominal execution time, and other factors. Adding randomization to these algorithms does not significantly improve their performance.
  - **Neural networks perform well but are too expensive.** For short nominal times and even tight bounds, a neural network approach was consistently near optimal. However, the compute costs are clearly too high for this approach to be practical.

- **Simple and effective mapping algorithm:** I developed a mapping algorithm that is practical and yet gives near optimal performance for short nominal times and tight bounds. Even for long nominal times, it performs better than any other algorithm we tested. However, it is sensitive to varying nominal times.

The thesis work will be focused on extending my methods and results to the full dynamic mapping problem. The contributions I expect from this work include:

- **Methodology for collecting activation tree traces from real applications:** As far as I know, no such methodology exists, and yet such traces are clearly needed for analysis and simulation. The overall structure of this methodology is already in place, as discussed in Section 9.

- **Methodology for collecting network traces:** Although networking researchers regularly collect traces to analyze, simulate, and improve *networks*, I want to collect traces that are useful for simulating the communication of applications on networks in order to improve the *applications*.

- **Activation tree traces:** These traces will let myself and others simulate and analyze real interactive applications. No collection of such traces exists at the present time, so such a collection may also be quite useful to other researchers.
- **Network traces:** These traces will let myself and others simulate realistic communication times. It may also be possible to use traces collected by networking researchers.
- **Analysis of activation tree traces:** I intend to analyze the activation tree traces that I collect to the same level of detail as I analyzed load traces in my current work. My analysis will be geared toward history-based prediction. I expect that one of the results of this analysis will be a model and methodology for generating activation trees with similar properties.
- **Analysis of network traces:** My analysis of the network traces will also be geared toward history-based prediction. I am not interested in long term aggregate properties of the network, but in short term properties on the scale of the computation and communication times of remote execution facilities.
- **Benchmarks:** A benchmark is a combination of an activation tree trace and some number of host load and network traces that can be used to simulate running a particular interactive application in a particular distributed computing environment. A family of such benchmarks would be very useful to researchers in comparing different dynamic mapping algorithms.
- **Communication time model:** The extended simulator will have to determine communication times given data sizes and network traces. I will develop and validate a model for this purpose. I am not sure yet whether I will limit myself to client/server communication (request-response between a pair of hosts) or attempt the many-to-one/one-to-many parallel communication one would expect in a optimized shared memory environment.
- **Extended simulation infrastructure:** Using the communication time model I will extend my simulator to support the full dynamic mapping problem. I will keep the simulator as simple and realistic as possible by directly using the traces I will have collected earlier. I intend to maintain the current "plug-and-play" nature of the simulator so that will be useful for future researchers as well as myself.
- **Mapping algorithms for full dynamic mapping problem:** The core contribution of this thesis will be mapping algorithms for the full dynamic mapping problem. I expect that this will involve a prediction algorithm for estimating the number of children of a given activation tree node and the distribution of nominal time among them, and an extension of the algorithm I developed for the simplified problem for mapping individual nodes.
- **Extensive evaluation of algorithms in simulator:** I will evaluate the algorithms I develop using my simulator on the benchmarks and other test sets. The goal is to do as much evaluation as possible in simulation. This seems reasonable given the simulator's trace-based nature, although it will be necessary for the evaluation to be extensive in order to avoid dependencies on particular traces. This extensive nature will also help to illustrate how each algorithm depends on the properties of the environment and program.
- **Validation of mapping algorithm in LDOS:** As a proof of concept, I will incorporate the best mapping algorithm I develop into the LDOS system. The goal here is to show that my approach is not limited to a simulation environment, but can be incorporated into a real system.

11 Related work

In this section, we discuss related work in applications, remote execution facilities, distributed soft real-time systems, and quality of service systems.

Applications

There is considerable interest in distributed interactive applications for visualizing and steering scientific computations, manipulating media, and playing games. The goal of the thesis work is to simplify building these sorts of of applications.

Scientific visualization involves the useful display of large, complex datasets, while computational steering lets the user use this display to focus a running physical simulation on areas of interest [35]. For example, the CAVE project [22] involves connecting immersive virtual environment simulators to remote physical simulations. CAVE has also proven to be a good environment for scientific and engineering collaboration [63]. Cumulus [33] is a library and run-time system for adding adding visualization and computational steering to simulations based on iterative operations on dense matrices. Tele-operation is related to computational, except that the remote physical simulation is replaced with an actual physical system. Tele-operation systems such as the one discussed in [56], involve operating robots by remote control over conventional networks.

While most current distributed multimedia systems involve communicating and synchronizing different audio and video streams, manipulating media on a distributed system in computationally significant ways is becoming increasingly important. For example, medical imaging systems [62], involve acquiring, processing, storing, and making medical images easily available to physicians no matter where they are. Next generation stream-oriented systems like the VuSystem [48] filter the streams in computationally intensive ways, such as extracting titling text from video streams. Image editing programs such as Photoshop [1] were also ripe for distributed systems because of the increasingly huge size of photographic images [18].

Games are becoming increasingly interesting applications for distributed systems. For example, the Department of Defense's DIS effort is trying to create large scale (100,000 users or more) simulated war games by connecting different kinds of simulators located all over the world [21]. At least three companies, Silicon Graphics, Zombie Entertainment, and Military Simulations, Inc., are working on extending the emerging DIS technology for civilian games, with an example of a DIS-based game being shown at SIGGRAPH '96. In addition to multiplayer games, we expect that single player games will become increasingly rely on computationally intensive physical simulations for realism. For example, the use of physically based acoustics and music [2, 41, 72] in games may not be far off.

Remote execution

Our machine model includes a remote execution facility, which allows a procedure call to be executed on any host in the system. The purpose of our work is to decide which host is best for a given call. Examples of remote execution facilities include remote procedure call systems, distributed shared memory mechanisms, and distributed object systems. Remote procedure call systems [12, 9], such as the one specified by the DCE [75] standard, provide a procedure call-like abstraction over network communication with a remote server. Distributed object systems [60, 16], such as CORBA [76, 58, 69], DCOM [14] and Java RMI [74], extend the RPC abstraction to objects by allowing the association of state with a group of procedures. Distributed shared memory systems such as Tempest [64], Shasta [66], and ThreadMarks [23] provide the appearance of a global address space shared by some number of private memory hosts. This is combined with a multithreading model where threads that may be assigned to different hosts communicate via shared variables. Although today's common remote execution facilities have relatively high overheads in the millisecond range [34], our experience is that this is due mostly to the latency of network protocol stacks and we expect that this latency will improve. With microsecond range application-to-application communication latencies such as the PAPERS network [38] can provide, the overhead of remote execution in a system like LDOS or CORBA could be within a couple of orders of magnitude of the overhead of a local call. [6] Such an improvement would drastically increase the fraction of procedures in a typical program that could be profitably executed remotely.

Dynamic load balancing

Dynamic load balancing involves assigning tasks to hosts at run-time or moving tasks from one host to another at run-time in order to maximize some performance metric. Common performance metrics include the throughput of the system, the mean execution time of tasks, or the actual running time of an application. In contrast, the goal of our work is to maximize the number of tasks (activation trees) that meet their time bounds. There are both operating system-centric and application-centric approaches to dynamic load balancing. Our work is wholehearted application-centric — we want to predict the behavior of the rest of the system in order to improve the performance of one specific application. However, we share an important question with both forms of dynamic load balancing: how can an individual host in a distributed system be aware and predict the behavior of the system as a whole in a scalable, practical way?

The goal of operating system-centric dynamic load balancing is usually to maximize the throughput of the distributed system as a whole. The idea is to schedule independent, sequential tasks on a group of hosts such that each host has approximately the same load. A distinction is sometimes drawn between load sharing [15] (also called load distribution and load leveling), which offers only a rough approximation to equalizing load across the hosts, and load balancing, which attempts more precision [25]. In either case, the distributed operating system is comprised of host-local schedulers which interact to implement transfer (should task be moved?) and location (where should task go?) policies [24]. In our work, we schedule dependent tasks (activation tree nodes) and do not balance loads across different hosts.

Early work in operating system-centric dynamic load balancing assumed simple, analytically tractable distributions for job length and other properties. Under these assumptions, simple heuristics and initial placement were found to be adequate [26, 25]. However, measurement studies [46] have cast doubt on these assumptions, and more recent work [37] argues strongly for process migration. This is an important result for our work, since it argues for dynamically mapping individual activation tree nodes (letting the computation migrate at procedure call points) as opposed to mapping the whole tree (initial placement of the whole computation.)

The goal of application-centric dynamic load balancing is to minimize the execution time of a single, typically parallel application. For example, in data parallel applications built in DOME [3], neighboring hosts periodically exchange measurements of execution time and redistribute their data accordingly. The system described in [70] uses a global approach where one centralized agent makes load balancing decisions based on execution times reported by all of the hosts. Jade [65] load balances unfolding task parallel computations by dynamically mapping task graph nodes to hosts to minimize communication and overall execution time. In contrast, our work dynamically maps activation tree nodes that are executed sequentially.

In order to be scalable, a dynamic load balancing system pursues its global policy by making local decisions based on limited or outdated knowledge of the system as a whole. Based on this limited knowledge, a local scheduler estimates the current state of the system and makes its decisions based on that estimate. By exploiting subtle properties of specific distributed systems or application workloads, some systems can put their limited knowledge to better use. In essence, this is what we intend to do for a specific class of applications. One example of exploiting such properties is Hailperin's thesis [36], which takes advantage

[6] The marshalling/unmarshalling and dispatch overhead of a call in LDOS is only about 100 times greater than that of a C++ virtual function call.

of the statistical periodicity of his sensor-driven applications. Our applications have aperiodically arriving activation trees. In [68], the authors present location policies that adapt to system load to avoid instability. Mehra and Wah [52] use comparator neural networks to predict current workload indices on remote hosts using outdated information about resource utilization. Mehra's thesis [51] describes a whole load balancing system based on automated strategy learning using comparator neural networks. We have also found some success with a neural networks approach. Indeed, it is interesting to note that machine learning [53] approaches are also finding success in related scheduling problems. For example, genetic algorithms have been successful in branch prediction [27] and scheduling loop level parallelism in multiprocessors [79].

Distributed soft real-time systems

A real-time system allows a programmer to specify a deadline for the execution of a one-time or periodic task. The bounds in our dynamic mapping problem correspond to these deadlines. However, while real-time systems typically provide some form of guarantees, our work is strictly best effort. Further, most real-time systems require control of the entire computing environment, either via resource reservation or priority-based scheduling. In contrast, we control only the mapping of activation tree nodes to hosts.

A hard real-time system is one which is only correct when every task in the system always meets its deadlines. These systems are necessary for some applications, but are very difficult to build, place many constraints on the hardware, and typically require knowledge of all tasks at design time. The rate-monotonic scheduling and earliest deadline first scheduling algorithms [49] can be used to produce feasible schedules for fixed sets of tasks with known characteristics. Distributed hard real-time systems have been built (eg, MARS [67]), but these tend to rely on specialized hardware. In contrast, our work offers only a best effort service, but the tasks and deadlines are discovered only as the program is executed.

A soft real-time system is one where it is permissible for some tasks to miss their deadlines. What it means to miss a deadline depends on the system. A common approach is is to generalize a deadline into a utility function of the duration since the task arrival time [43]. The system then attempts to maximize the collective or individual utilities of tasks in the system. The functional form of a hard deadline in this kind of system is a unit step function. While the general case of this scheduling/optimization problem is quite complex, by constraining the form of the utility function, tractable specialized scheduling algorithms can be developed. The common underlying mechanism in soft real-time is a fixed priority scheduler combined with priority inheritance to address priority inversion. In a fixed priority scheduler, each task is assigned a fixed priority and the scheduler always runs the highest priority task that is ready to run. Unfortunately, very few general purpose operating systems actually implement fixed priority scheduling. In contrast, we have a very simple performance metric (the percentage of activation trees that meet their deadlines) and our work does not require specific kinds of schedulers or any real control over the system other than a remote execution facility and measurement facility.

Achieving soft real-time goals in a distributed system is very difficult since the system is much larger and more complex. One approach is to create a notion of a global priority, require that all resources in the system be scheduled by fixed priority schedulers that respect the fixed priority, and require that all communication delays be bounded. This is the approach taken in [78] and in proposals to the Real-time CORBA standardization effort [59, 20]. Another approach is to implement resource schedulers that provide reservation mechanisms, such as in SONIC [61]. In contrast, we make no attempt to control the system in order to meet our goals. Rather, we monitor and predict the system in order to adapt the application's behavior (the mapping of activation tree nodes) to the system and meet our bounds.

As in dynamic load-balancing, a key challenge in distributed soft real-time systems is how to scalably coordinate the actions of many local schedulers. If a task is submitted at a host where it cannot meet its deadline, we would like to move it to a host where it is likely to be able to. However, that host's knowledge of other hosts is limited and out of date. Bestavros and Spartiotis suggest a scheme in which simple stochastic models of other hosts' load conditions are formed based on infrequent, opportunistic information exchanges and task mappings are chosen probabilistically from among hosts that qualify according to the models [11]. Bestavros extends this work by arguing that load balancing is actually detrimental to real-time performance and develops an alternative which intentionally tries to keep loads out of balance and makes task mapping decisions based on the underlying load profile it is trying to maintain [10]. We suspect that Bestavros has more success with simple stochastic models than we do because he is modeling the remote schedulers that are a part of his system and are therefore a known quantity. In addition to load balancing, Hailperin's thesis [36] also takes advantage of the statistical periodicity of his sensor-driven applications to help place and migrate objects so that method invocations meet their deadlines. We must deal with aperiodically arriving activation trees.

Quality of service

Quality of service, or QoS, is a fairly generic, but nonetheless widely used term to describe providing a more predictable environment for applications. In this sense, our work can be seen as a QoS service for a specific class of applications. In the networking community, QoS has meant providing deterministic and statistical guarantees of the bandwidth and latency of connections for media applications [29, 45]. Tenet [30], RSVP [13] and the ATM CBR and VBR service classes are examples of network QoS systems. Although we make no assumptions about the network, one could imagine using such mechanisms to simplify the dynamic mapping problem by making the communication costs known a priori.

Recently, interest has focused on how to expose network and host QoS features to the application programmer in an understandable fashion. For example, work at BBN [81, 80] is considering how to express QoS requirements and guarantees for distributed objects in a collaboration system and similar applications. The challenge is in translating between the object level and the network level and in providing a clean way for the programmer to identify different regions of operation and when to switch between them. Another example of translation and adaptation involves the Qual language being developed at Columbia [31]. Qual is a language for expressing the quality requirements of program modules. These specifications can then be compiled into a monitoring agent which switches between different regions of operation. Compilable specification languages to support binding modules and services have also been developed in the parallel computing community. DURRA [6] is a well known example. In our work, the bounds placed on the execution time of an activation tree could be seen as a QoS specification.

In mobile computing the QoS challenge is to detect changing wireless network conditions quickly and adapt to them gracefully. For example, the Odyssey system [57] provides the application with typed data streams (eg, video) and switches between different quality levels (data rates) as it detects changing available network bandwidth. When quality strays outside of the range that the application registered initially, an upcall is made to the application, which decides the course of action.

References

[1] ADOBE CORPORATION. *Adobe Photoshop 4.0 User's Guide*, 1997.

[2] ALLEN, J. B., AND BERKLEY, D. A. Method for simulating small-room acoustics. *Journal of the Acoustical Society of America 65*, 4 (April 1979), 943–950.

[3] ARABE, J., BEGUELIN, A., LOWEKAMP, B., E. SELIGMAN, M. S., AND STEPHAN, P. Dome: Parallel programming in a heterogeneous multi-user environment. Tech. Rep. CMU-CS-95-137, Carnegie Mellon University, School of Computer Science, April 1995.

[4] BAO, H., BIELAK, J., GHATTAS, O., KALLIVOKAS, L. F., O'HALLARON, D. R., SHEWCHUK, J. R., AND XU, J. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Computer Methods in Applied Mechanics and Engineering 152*, 1–2 (Jan. 1998), 85–102.

[5] BAO, H., BIELAK, J., GHATTAS, O., O'HALLARON, D. R., KALLIVOKAS, L. F., SHEWCHUK, J. R., AND XU, J. Earthquake Ground Motion Modeling on Parallel Computers. In *Supercomputing '96* (Pittsburgh, Pennsylvania, Nov. 1996).

[6] BARBACCI, M. R., AND WING, J. M. DURRA: A task-level description language. In *Proceedings of The International Conference on Parallel Processing* (August 1987), pp. 370–376.

[7] BASSINGTHWAIGHTE, J. B., BEARD, D. A., PERCIVAL, D. B., AND RAYMOND, G. M. Fractal structures and processes. In *Chaos and the Changing Nature of Science and Medicine: An Introduction* (April 1995), D. E. Herbert, Ed., no. 376 in AIP Conference Proceedings, American Institute of Physics, pp. 54–79.

[8] BEGAULT, D. R. *3-D Sound For Virtual Reality and Multimedia*. AP Professional, 1994.

[9] BERSHAD, B. N., ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H. M. Lightweight remote procedure call. *ACM Transactions on Computer Systems 8*, 1 (February 1990), 37–55.

[10] BESTAVROS, A. Load profiling: A methodology for scheduling real-time tasks in a distributed system. In *Proceedings of ICDCS '97* (May 1997).

[11] BESTAVROS, A., AND SPARTIOTIS, D. Probabilistic job scheduling for distributed real-time applications. In *Proceedings of the First IEEE Workshop on Real-Time Applications* (May 1993).

[12] BIRREL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39–59.

[13] BRADEN, R., ZHANG, L., BERSON, S., HERZOG, S., AND JAMIN, S. Resource ReSerVation protocol (RSVP) – version 1 functional specification. Internet RFC 2205, September 1997. ftp://ftp.isi.edu/in-notes/rfc2205.txt.

[14] BROWN, N., AND KINDEL, C. Distributed component object model protocol – dcom/1.0. Tech. rep., Microsoft, May 1996. http://ds1.internic.net/internet-drafts/draft-brown-dcom-v1-spec-00.txt.

[15] BUBENDORFER, K. P. Resource based policies for load distribution. Master's thesis, Victoria University of Wellington, August 1996.

[16] CHIN, R., AND CHANSON, S. Distributed object-based programming systems. *ACM Computing Surveys 23*, 1 (March 1991), 91–124.

[17] DEWITT, T., GROSS, T., LOWEKAMP, B., MILLER, N., STEENKISTE, P., SUBHLOK, J., AND SUTHERLAND, D. ReMoS: A resource monitoring system for network aware applications. Tech. Rep. CMU-CS-97-194, School of Computer Science, Carnegie Mellon University, December 1997.

[18] DINDA, P. A. Network personal computing for world simulation. Tech. Rep. Working Paper, Carnegie Mellon University School of Computer Science, February 1996.

[19] DINDA., P. A., GARCIA, B. M., AND LEUNG, K. S. Traffic characterization of parallel programs. http://www.cs.cmu.edu/afs/cs/project/iwarp/member/pdinda/PAPERS/traf_char/report.ps, May 1995.

[20] DIPIPPO, L. C., GINIS, R., SQUADRITO, M., WOHLEVER, S., WOLFE, V. F., ZYKH, I., AND JOHNSTON, R. Expressing and enforcing timing constraints in a CORBA environment. Tech. Rep. 97-252, Department of Computer Science, University of Rhode Island, February 1997.

[21] DIS STEERING COMMITTEE. The dis vision: A map to the future of distributed simulation. Tech. Rep. Version 1, DIS Steering Committee, May 1994.

[22] DISZ, T. L., PAPKA, M. E., PELLEGRINO, M., AND STEVENS, R. Sharing visualization experiences among remote virtual environments. In *Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualization* (1995), pp. 217–237.

[23] DWARKADAS, S., COX, A., AND ZWAENEPOEL, W. An integrated compile-time/run-time software distributed shared memory system. In *Proc. Sixth Intl. Conf. on Architectural Support for Prog. Languages and Operating Systems (ASPLOS VI)* (Boston, Oct 1996), ACM.

[24] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering SE-12*, 5 (May 1986), 662–675.

[25] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation 6* (March 1986), 53–68.

[26] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. The limited performance benefits of migrating active processes for load sharing. In *SIGMETRICS '88* (May 1988), pp. 63–72.

[27] EMER, J., AND GLOY, N. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (June 1997).

[28] ENDO, Y., WANG, Z., CHEN, J. B., AND SELTZER, M. Using latency to evaluate interactive system performance. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (1996).

[29] FERRARI, D. Client requirements for real-time communication services. *IEEE Communications Magazine 11*, 11 (November 1990), 65–72.

[30] FERRARI, D., BANERJEA, A., AND ZHANG, H. Network support for multimedia - a discussion of the tenet approach. *Computer Networks and ISDN Systems 26*, 10 (July 1994), 1167–1180.

[31] FLORISSI, P. G. S. *QuAL: Quality Assurance Language (Thesis Proposal)*. PhD thesis, Columbia University Computer Science Department, March 1994.

[32] GARRETT, M., AND WILLINGER, W. Analysis, modeling and genreation of self-similar VBR video traffic. In *Proceedings of SIGCOMM '94* (London, September 1994).

[33] GEIST, G. A., KOHL, J. A., AND PAPADOPOULOS, P. M. CUMULVS: Providing fault-tolerance, visualization, and steering of parallel applications. Tech. rep., Oak Ridge National Laboratory, 1996.

[34] GOKHALE, A., AND SCHMIDT, D. C. Measuring the performance of communication middleware on high-speed networks. In *Proceedings of SIGCOMM 96* (August 1996). Extended version available as TR.

[35] HAGGEN, H., MUELLER, H., AND NIELSON, G. M., Eds. *Focus On Scientific Visualization*. Springer-Verlag, 1993.

[36] HAILPERIN, M. *Load Balancing using Time Series Analysis For Soft Real-Time Systems with Statistically Periodic Loads*. PhD thesis, Stanford University, December 1993.

[37] HARCHOL-BALTER, M., AND DOWNEY, A. B. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of ACM SIGMETRICS '96* (May 1996), pp. 13–24.

[38] HOARE, R., DIETZ, H. G., MATTOX, T., AND KIM, S. Bitwise aggregate networks. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing* (October 1996).

[39] HOLLINGSWORTH, J. K., AND BUCK, B. *DynInstAPI Programmer's Guide Release 1.0*, July 1997. http://www.cs.umd.edu/ hollings/dyninstAPI/dyninstUserGuide.pdf.

[40] HOLLINGSWORTH, J. K., MILLER, B. P., AND CARGILLE, J. Dynamic program intstrumentation for scalable perforamnce tools. In *Proceedings of SHPCC '94* (May 1994).

[41] HUOPANIEMI, J., KARJALAINEN, M., VAELIMAEKI, V., AND HOUTILAINEN, T. Virtual intstruments in virtual rooms — a real-time binaural room simulation environment for physical models of musical instruments. In *Proceedings of the International Computer Music Conference (ICMC '94)* (September 1994).

[42] HURST, H. E. Long-term storage capacity of reservoirs. *Transactions of the American Society of Civil Engineers 116* (1951), 770–808.

[43] JENSEN, E. D. A real-time manifesto. http://www.realtime-os.com, 1996.

[44] KAPLAN, J. A., AND NELSON, M. L. A comparison of queueing, cluster, and distributed computing systems. Tech. Rep. NASA TM 109025 (Revision 1), NASA Langley Research Center, June 1994.

[45] KUROSE, J. Open issues and challenges in providing quality of service guarantees in high speed networks. *ACM Computer Communication Review 23*, 1 (January 1993), 6–15.

[46] LELAND, W. E., AND OTT, T. J. Load-balancing heuristics and process behavior. In *Proceedings of Performance and ACM SIGMETRICS* (1986), vol. 14, pp. 54–69.

[47] LELAND, W. E., TAQQU, M. S., WILLINGER, W., AND WILSON, D. V. On the self-similar nature of ethernet traffic. In *Proceedings of ACM SIGCOMM '93* (September 1993).

[48] LINDBLAD, C. J., WETHERALL, D. J., AND TENNENHOUSE, D. L. The VuSystem: A programming system for visual processing of digital video. In *Proceedings of the ACM Multimedia 94* (October 1994), ACM.

[49] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (January 1973), 46–61.

[50] MALCEVIC, I. Large-scale unstructured mesh shape optimization on parallel computers. Master's thesis, School of Civil and Environmental Engineering, Pittsburgh, PA, 1997.

[51] MEHRA, P. *Automated Learning of Load-Balancing Strategies for a Distributed Computer System*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[52] MEHRA, P., AND WAH, B. W. Automated learning of workload measures for load balancing on a distributed system. In *Proceedings of IPPS '93* (August 1993), CRC Press, pp. III–263–III–270.

[53] MITCHELL, T. M. *Machine Learning*. McGraw-Hill, 1997.

[54] MOON, F. C. *Chaotic and Fractal Dynamics: An Introduction for Applied Scientists and Engineers*. John Wiley and Sons, Inc., 1992.

[55] MORIN, P. R. The impact of self-similarity on network performance analysis. Tech. Rep. Computer Science 95.495, Carleton University, December 1995.

[56] NAHRSTEDT, K., AND SMITH, J. M. An application-driven approach to networked multimedia systems. In *Proceedings of the 18th Annual Conference on Local Area Computer Networks* (1993), IEEE, pp. 361–368.

[57] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (1997). To Appear.

[58] OBJECT MANAGEMENT GROUP. The common object request broker: Architecture and specification. Tech. Rep. Version 2.0, Object Management Group, July 1995.

[59] OBJECT MANAGEMENT GROUP. Realtime corba: A white paper. http://www.omg.org, December 1996. In Progess.

[60] ORFALI, R., HARKEY, D., AND EDWARDS, J. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, Inc., New York, New York, 1996.

[61] POLZE, A., WERNER, M., AND FOHLER, G. Predictable network computing. Tech. rep., Humboldt University of Berlin, 1996.

[62] PYARALI, I., HARRISON, T. H., AND SCHMIDT, D. C. Design and performance of an object-oriented framework for high-speed electronic medical imaging. *Usenix Computing Systems Journal 9*, 3 (November/December 1996). earlier version appeared in the Usenix COOTS Conference, June, 1996.

[63] RANTZAU, D., LANG, U., LANG, R., NEBEL, H., WIERSE, A., AND RUEHLE, R. Collaborative and interactive visualization in a distributed high performance software environment. In *Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualization* (1995), pp. 207–216.

[64] REINHARDT, S., LARUS, J., AND WOOD, D. Tempest and typhoon: User-level shared memory. In *Proceedings of ISCA '94* (April 1994), ACM, pp. 325 – 336.

[65] RINARD, M., SCALES, D., AND LAM, M. Jade: A high-level machine-independent language for parallel programming. *IEEE Computer 26*, 6 (June 1993), 28–38.

[66] SCALES, D. J., GHARACHORLOO, K., AND THEKKATH, C. A. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996).

[67] SCHWABL, W., REISINGER, J., AND GRUENSTEIDL, G. A survey of MARS. Tech. Rep. 16/89, Technische Universitaet Wien, October 1989.

[68] SHIVARATRI, N. G., AND KRUEGER, P. Two adaptive location policies for global scheduling algorithms. In *Proceedings of ICDCS '90* (May 1990), pp. 502–509.

[69] SIEGAL, J. *CORBA Fundamentals and Programming*. John Wiley and Sons, Inc., 1996.

[70] SIEGELL, B., AND STEENKISTE, P. Automatic generation of parallel programs with dynamic load balancing. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing* (August 1994).

[71] SMITH, G. D. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Clarendon Press, 1985.

[72] SMITH, J. O. Physical modeling synthesis update. *Computer Music Journal 20*, 2 (Summer 1996), 44–56.

[73] STEENKISTE, P., FISHER, A., AND ZHANG, H. Darwin: Resource management for application-aware networks. Tech. rep., School of Computer Science, Carnegie Mellon University, 1997. Draft overview paper (http://www.cs.cmu.edu/afs/cs/usr/prs/WWW/darwin.ps).

[74] SUN MICROSYSTEMS, INC. Java remote method invocation specification. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html, 1997.

[75] THE OPEN GROUP. *DCE 1.2.2: Introduction to OSF DCE*. The Open Group, September 1997. http://www.opengroup.org/pubs/catalog/f201.htm.

[76] VINOSKI, S. Distributed object computing with CORBA. *C++ Report* (July/August 1993).

[77] WILLINGER, W., MURAD S, T., SHERMAN, R., AND WILSON, D. V. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. In *Proceedings of ACM SIGCOMM '95* (1995), pp. 100–113.

[78] WOLFE, V. F., BLACK, J. K., THURAISINGHAM, B., AND KRUPP, P. Real-time method invocations in distributed environments. Tech. Rep. 95-244, Department of Computer Science, University of Rhode Island, January 1996.

[79] YUE, K. K., AND LILJA, D. J. Designing multiprocessor scheduling algorithms using a distributed genetic algorithm system. Tech. Rep. HPPC-96-03, University of Minnesota High-Performance Parallel Computing Research Group, May 1996.

[80] ZINKY, J. A., BAKKEN, D. E., AND SCHANTZ, R. Overview of quality of service for distributed objects. In *Proceedings of the IEEE Dual Use Technologies Conference* (May 1995).

[81] ZINKY, J. A., BAKKEN, D. E., AND SCHANTZ, R. E. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems* (January 1997).