Using Cause-Effect Analysis to Understand the Performance of Distributed Programs*

Wagner Meira Jr.
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, MG, Brazil
{meira,virgilio}@dcc.ufmg.br

Thomas J. LeBlanc
Virgílio A. F. Almeida
Department of Computer Science
University of Rochester
Rochester, NY, USA
leblanc@cs.rochester.edu

Abstract

Understanding the performance of distributed programs can be very difficult, since a program's performance depends on characteristics of the application, the underlying hardware, the software environment, and interactions among all three. In this paper we present cause-effect analysis (CEA), a general approach to understanding distributed program performance that facilitates performance analysis, tuning, and prediction. Using detailed program traces gathered at execution time as input, CEA automatically generates explanations for important performance phenomena, identifying code segments that are responsible for the occurrence of the phenomena.

We illustrate our approach by describing CEA techniques for three classes of overheads in distributed programs: contention, synchronization, and communication. Using the explanations produced by CEA, we are able to understand and minimize common performance problems in real applications including load imbalance, false sharing, and resource contention.

1 Introduction

In order to understand the performance of a distributed program, we must be able to explain performance phenomena in terms of the decisions a programmer makes while creating the program. An *explanation* for a particular phenomenon (such as excessive time spent waiting at a barrier at the end of a loop) is the set of statements, implementation decisions (such as the loop scheduling algorithm or the data layout scheme), or architectural characteristics that produced the phenomenon. At present, explanations for program behavior are inferred manually by the programmer, usually based on an analysis of the source code and some dynamic measurements. This type of manual analysis is complex, error-prone, and time-consuming.

*This research was supported by an NSF grant CCR-9510173, an NSF CISE Institutional Infrastructure Grant CDA-9401142, and an equipment grant from Digital Equipment Corporation's External Research Program. Wagner Meira Jr. was supported by CNPq/Brazil grant 200.862/93-6.

SPDT 98 Welches OR USA

Events whose duration varies significantly during execution are especially difficult to analyze manually. Any event whose duration is determined by interactions among processes or the duration of other events we will call a non-deterministic duration (NDD) event. An example of an NDD event is a page fault that occurs in a DSM system. The duration of the page fault is a function of the accesses performed by other processes since the page was last acquired by the faulting process. To understand why a page fault occurs, and why so much time is spent handling the fault, we must examine the interactions between all processes that share access to the page.

Another example of an NDD event is a process arrival at a barrier. The duration of the delay experienced by a process at a barrier depends on the execution path of the last process to reach the barrier. To understand why processes waste excessive amounts of time waiting at a barrier, we must focus on this execution path as the source of the delays.

Discovering explanations for NDD events is difficult however, both because there may be multiple causes for each performance phenomenon and because causes and effects may be spread out in terms of code location and time of occurrence. Although there are many tools that can identify the existence of performance problems (such as an excessive amount of time spent synchronizing during execution), there are no tools that automatically relate the durations of performance phenomena to the underlying causes.

In this paper we present a novel approach to understanding the duration of NDD events: cause-effect analysis (CEA) [12]. This approach is a generalization of our previous work on performance understanding [13, 14], which uses runtime traces to generate execution profiles that guide the user in identifying the location and cost associated with various parallel program overheads. CEA augments our profile-based analysis by automatically linking instances of performance degradation with the source code that must be tuned in order to address the performance problem. Both profiles and CEA techniques are implemented within Carnival [12], an integrated framework for performance understanding that automates most of the tasks related to performance understanding, including instrumentation, monitoring, analysis, visualization, and generation of explanations for performance phenomena based on CEA.

This paper is organized as follows. The next section presents a brief overview of Carnival. Section 3 describes the general idea behind cause-effect analysis and presents three specific analysis techniques that we have developed. We present some practical examples that utilize these techniques in Section 4. The last two sections summarize related

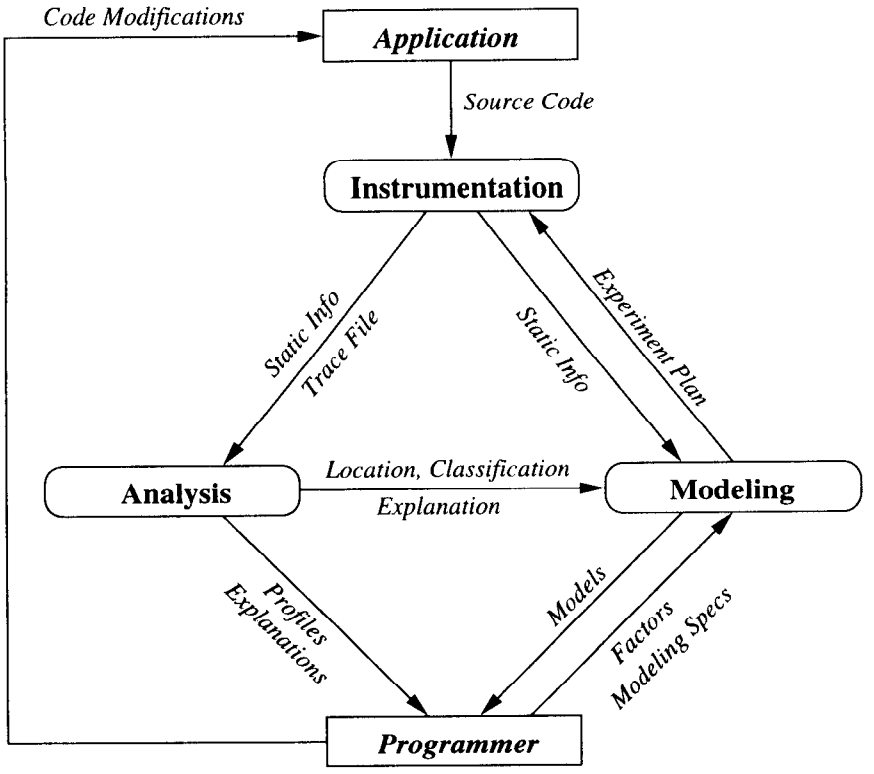

Figure 1: Carnival: Components

work and our conclusions.

2 Using Carnival for Performance Understanding

Carnival is a framework for performance understanding that supports four levels of performance description: (i) it *characterizes* performance at a coarse grain using data such as sequential and parallel execution times, speedup, and efficiency; (ii) it *classifies* the program's execution time by dividing it among all sources of parallel overhead, as well as normal computation; (iii) it *identifies* the code locations where parallel overheads and computation costs arise; and (iv) it *predicts* the effects of implementation decisions on program performance using analytic models that are automatically generated from execution information.

Programmers may use Carnival for either analyzing individual executions or investigating trends across multiple executions. There are three main components in Carnival(as depicted in Figure 1):

Instrumentation: Instrumentation comprises tools and libraries that are used to acquire detailed runtime traces of an application during its execution. Instrumentation is the only platform-dependent component, so Carnival can be ported easily to new architectures.

Analysis: The analysis component consists of tools that analyze trace files, produce execution profiles, and automatically generate a graphical interface that allows these profiles to be visualized. Figure 2 shows a *Profile Map* produced by Carnival . Each line represents a scope from the application, which can be a loop, a procedure, or a block of code. Each scope has a greyscale bar that indicates the cumulative execution time (across all processors) spent in that scope. The colors in the horizontal bar at the top of each scope describe a breakdown of the execution time spent in the scope into categories. By using this profile map, the programmer can easily identify the scopes that dominate the execution time of the application.

Modeling: Carnival incorporates the Lost Cycles Toolkit (LCT) [5], which automates the process of constructing a performance model for a parallel application by integrating empirical model-building techniques from statistics with measurement and modeling techniques for parallel programs.

Carnival differs from other performance tools by integrating five features that are essential for understanding performance:

Static Information: The static information in Carnival is the source code of the application and its structure. The source code is organized as a hierarchy of nested scopes, according to the semantic structure of the application. This static information is exploited in dynamic data collection, visual interface generation, and modeling.

Dynamic Information: Carnival provides an instrumentation and tracing library for acquisition of execution-time performance data. Each call to a library function generates an event that registers a processor identifier, scope, processing category, and time of occurrence. These events are stored in a trace file for further processing.

Hierarchical Abstraction of Performance Data: Performance information is organized hierarchically according to the source code structure in Carnival, allowing analysis and modeling at several levels of abstraction.

Integration of Dynamic and Static Data: Carnival integrates dynamic and static information by associating with each dynamic event the corresponding code segment and processing category.

Support for Generating Models Automatically: Carnival automates experimental design and multivariate modeling. It also provides feedback to the programmer on the quality of the resulting model, and allows the programmer to trace inaccuracies back to the component models and performance data.

Carnival comprises about 50000 lines of C and Tcl/Tk code. We have implemented program tracing for message-passing programs on the IBM SP2 and shared-memory programs on the SGI Challenge. We also have implementations for Treadmarks [2] and PIOUS [16] on clusters of DEC Alphas connected by the DEC Memory Channel, PVM programs on a network of workstations, and Squid cache proxies [6]. More than two thirds of the Carnival code is devoted to the user interface, profile generation, and modeling support. Of the 15000 lines devoted to cause-effect analysis techniques, roughly one half are common to all the techniques (and are encapsulated in a library), with the remaining 7500 lines divided roughly equally among the three techniques we have implemented.

3 Cause-Effect Analysis

Cause-effect analysis is an automated technique for generating explanations for non-deterministic duration events (NDD). CEA exploits knowledge about the causal relationships between execution events [9], but differs from previous work in one key aspect: we are concerned primarily with the duration of events, and not simply with their occurrence. Under CEA, an event a causally affects an event b when the duration of a affects the duration of b. From this definition, it follows that cause-effect analysis subsumes other causality-based approaches, since a non-zero duration represents an occurrence.

CEA techniques are based on *Weighted Causality Graphs* (WCG) [12], which abstract the execution of distributed programs. Distributed programs are a collection of processes that reside on multiple processors. For the purposes of exposition, we assume that processors are homogeneous, the variance among their clock rates is negligible and bounded, and all processes start simultaneously. (We can apply a variety of techniques to relax these assumptions.) We abstract the execution of each process as a sequence of events, where each event corresponds to the contiguous execution of one or more instructions by the process. The granularity of an event ranges from single instructions to function calls, depending on the level of detail needed for program analysis and the instrumentation overhead introduced during execution. Each event has four attributes: code location, processor, type, and duration (which is always non-zero).

A weighted causality graph is an acyclic directed graph, where the vertices are execution events and the edges are causal relationships among those events. Each vertex in a WCG has a weight, which is equal to the duration of the corresponding event. There are two types of edges in WCGs. Intra–processor edges connect vertices that represent consecutive events on a single processor. Inter–processor edges connect synchronization or communication events on one processor with the corresponding event on another processor.

The transition between consecutive events on the same processor occurs instantaneously. However, the transition between consecutive events that occur on different processors, such as a send operation and the corresponding receive operation, is rarely instantaneous, due to transmission delays and other transient effects. To capture these delays in the WCG, during instrumentation we measure the time at which the send operation completes on one processor and the time at which a message is received on the other processor. We then post-process the trace and insert a synchronization event of the proper duration before the receive event (see [12] for details). Although this technique produces an approximation for transmission durations, any inaccuracies are eventually accounted for in the WCG, usually in a subsequent synchronization interval.

The time spent during execution by a processor is equal to the sum of the weights of the vertices for that processor in the WCG. Since there is a duration associated with each event in the WCG, and all processors start simultaneously, we can calculate a starting and ending time for each event in the WCG. All CEA techniques assume as input a WCG with these properties; in our implementation, we use the runtime traces gathered by Carnivalto construct the WCG.

From the perspective of a process P, we divide events into three groups (as depicted in Figure 3): (1) *useful computation* associated with the problem being solved by P; (2) *unrelated computation* performed by other processes that share the processor with P; and (3) *excess computation* performed by P that represents the overhead of parallelization and distribution. These groups of events characterize three possible sources of performance degradation in a distributed program: (1) contention – the execution of a process is delayed because it is contending with another process for the same shared resource; (2) synchronization – the execution of a process is delayed because it is idle waiting for another process at a synchronization operation; and (3) parallel overheads – the execution of a process takes longer than its sequential counterpart because it is performing additional operations, such as process creation or communication, that are needed for the parallel execution. These three sources of performance degradation are depicted in Figure 4, where we observe how they affect the execution of a simple program that consists of three events (as depicted in the sequence of events labeled "Normal Execution").

Different phenomena may require different analysis techniques, but all CEA techniques are designed and implemented as follows:

Identify Phenomena: We determine the execution events that characterize the occurrence of the phenomena of interest. For example, a process entering a barrier is an event that signals the onset of synchronization overhead, whose duration will vary depending on other events in the execution.

Determine Possible Causes: We determine the execution events that must be considered as a possible explanation for the duration of the events of interest. For example, the duration of a page fault event is independent of any preceding computation events (assuming those events did not themselves result in a coherence operation).

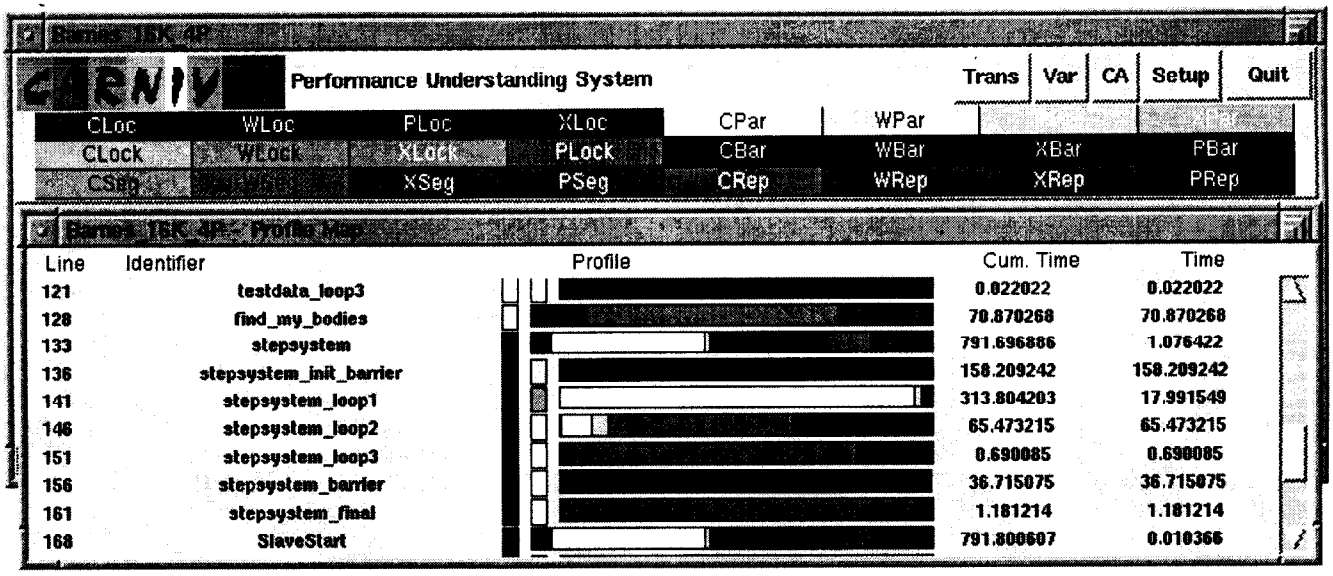

Figure 2: Code Profile from Barnes

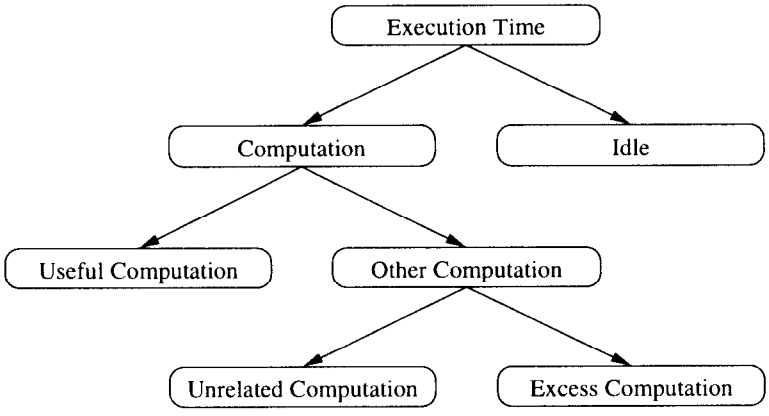

Figure 3: Decomposition of process execution time

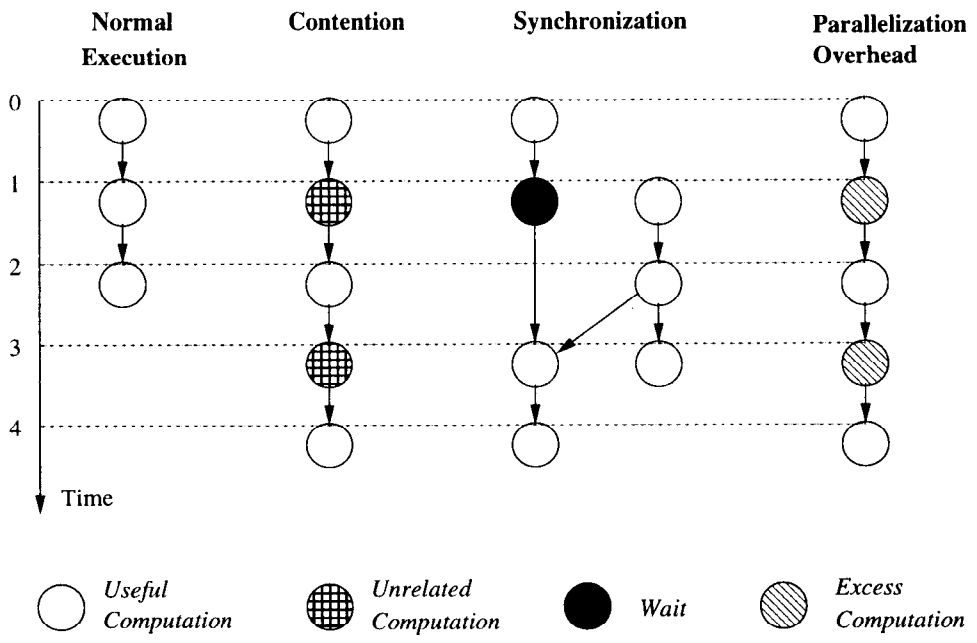

Figure 4: Sources of performance degradation in weighted causality graphs

Derive Explanations: Use the properties of the WCG and the phenomena of interest to isolate the set of events that must be considered in deriving an explanation. For example, when attempting to explain why a processor waited for another in a barrier, we need not analyze any events that happened prior to the last time these processors synchronized, since those events could not possibly affect the observed waiting time.

Simplify explanations: Design and implement routines to detect and automatically eliminate redundancies in explanations. For example, if two processes executed the same code for the same amount of time, the corresponding events cannot (by themselves) cause one processor to subsequently wait for another.

Merge similar explanations into characterizations: Determine any explanation attributes that can be abstracted when creating general characterizations of a phenomena. For example, in SPMD (Single Program Multiple Data) programs, we can abstract the processor identifier, because all processors execute the same code.

Generate visualization: Use Tcl/Tk [17] to implement the GUI that allows programmers to visualize both characterizations for performance phenomena and execution-time profiles.

In the following sections we describe three CEA techniques we developed and scenarios where these techniques can be employed for performance tuning.

3.1 Contention Analysis

Contention is a common source of performance degradation in distributed programs. Contention arises when multiple clients make requests to the same shared resource simultaneously. Client requests for data from a parallel file system or from a WWW cache proxy hierarchy are examples of operations whose duration may be affected by contention.

Contention analysis is a CEA technique that identifies requests that are contending for a shared resource and quantifies the impact of each source of contention on the overall performance of the distributed application. Contention analysis exploits the property of WCGs that all paths between two events have the same duration. The two events of interest are the request for a resource by a process and the granting of that resource by the server. The two execution paths of interest are the events performed by the client and server processes between these two events; any event performed by the server on behalf of another client contributes to the contention delay experienced by the original client.

To implement contention analysis, we first execute the program and collect traces of events. We then modify the trace file to adjust the durations of three classes of events (request resource, wait, and receive resource) so as to account for transmission delays (as described above). After adjusting the trace file, we build the WCG, including events on both the client and server processor.

The next step is to generate explanations for the durations of the various requests. Each explanation includes the amount of time spent satisfying the request, serving other requests, and internal server bookkeeping. For each event associated with an interfering request, the explanation describes its attributes (e.g., code location, type of operation, server) and its duration.

After generating an explanation for the duration of each individual request, we combine explanations into characterizations for classes of requests by merging the explanations for each request in that class. For example, a single characterization can summarize the sources of contention for all requests generated by a particular source code statement. We then generate a visualization of these characterization, which the programmer can use to discover interfering requests, and to quantify the impact of interference on each request's duration.

Contention characterizations can be used to pinpoint common resource allocation problems, such as load imbalance among parallel file servers. By analyzing these characterizations, programmers can detect resource-related operations that need to be optimized. and determine how best to redistribute resources.

3.2 Synchronization Analysis

Waiting time arises whenever a process is idle waiting for another process at a synchronization point. Waiting time can be reduced either by optimizing the code executed by the process causing the delay, or by reordering code segments on the delayed process, overlapping waiting time with computation. Synchronization analysis identifies the code segments that are responsible for the occurrence of waiting time, and quantifies the contribution of each code segment to the delay.

Synchronization analysis is based on the fact that two processes that perform *exactly* the same operations will experience no delays when they synchronize; it is the *differences* in their execution paths that introduce delays during synchronization. Since all execution paths in a WCG between two consecutive synchronization events have the same duration, the wait event executed by one processor must be the same duration as some set of computation events performed by the other processor. We generate an explanation for each wait event in the execution by traversing the WCG back up to the last synchronization event between the same two processes and comparing the two execution paths going forward. We identify any differences in the execution paths as the cause of the subsequent synchronization delay.

As with contention analysis, we can group explanations for a class of events into a single characterization, which summarizes the explanations for all events in the class. Thus, a single characterization can be used to explain all instances of waiting that arise from a single source code statement (e.g., a barrier). By analyzing these characterizations, the programmer can identify code segments that need to be optimized or reordered, so as to reduce overall waiting time.

3.3 Communication Analysis

Communication is a major source of performance degradation in distributed programs, since it always represents overhead introduced by the distributed implementation. Communication analysis (CA) exposes execution-time relationships that explain the duration of communication operations. In DSM systems, for instance, where the duration of a page fault is a function of preceding events (such as writes to the page or an invalidation of the page), these relationships capture the chain of events that result in the migration of a page across processors.

We have implemented a CEA technique that explains the cause and durations of page faults in a DSM system. (Similar ideas can be used to explain other forms of communication in distributed programs.) Our implementation of communication analysis exploits a property of the release consistency memory model used in Treadmarks [2]: multiple modifications to a page are grouped together and sent to other processors only once (usually at a synchronization point), and therefore we do not need to examine every individual page reference to understand the cause of a fault or its duration.

Our implementation of communication analysis explains the durations of remote requests (page faults) by analyzing traces of program executions that contain a record of every page fault and synchronization operation, with a global timestamp for each. Each page fault records the source code line that generated the fault, the nature of the fault (read or write), and the page number. Each synchronization operation records the list of pages that were invalidated as part of the operation. From the trace file, we build a WCG for each page. The vertices in the graph represent page faults (and their cause), and edges in the graph represent causal relationships. There is an edge between two vertices if the write fault explained by one vertex generates a write notice that is a cause for the fault in the other vertex. We assign weights to the edges according to the frequency that the edge is traversed. Also, each vertex is assigned an access pattern that reflects how the page is accessed. The access patterns are: multiple-producer-single-consumer, multiple-producer-multiple-consumer, single-producer-single-consumer, single-producer-multiple-consumer, cold start, and migratory.

From the WCG, the *cause* of each remote request is determined automatically, where a cause is the invalidation that preceded the page fault, and the write faults that generated the write notices at the synchronization point. As with the other CEA techniques, explanations are merged into characterizations, which summarize page fault behavior for a single page over the course of the execution or for multiple pages with similar behavior. With this information, the programmer can identify the source code, data structures, and access patterns that result in page requests, and thereby discover optimizations in data layout or scheduling to improve performance.

4 Examples

In this section, we describe how cause-effect analysis techniques can be used to understand and improve the performance of applications.

4.1 Contention in Parallel File Systems

The PIOUS file system [16] provides transparent access to data that is striped across multiple disk servers. In PIOUS, every file operation is a transaction, and strict two-phase locking is used to ensure sequential consistency. When a process issues a file operation, messages requesting the operation are sent to the appropriate disk servers, which perform the operation and return a result. The results are collected and, assuming all of them indicate successful completion of the operation, a *commit* message is sent to each participating disk server; otherwise, an *abort* message is sent. Commit messages cause the disk servers to make the changes permanent, while abort messages restore the previous state, after which the operation is typically tried again.

In the context of parallel file systems, contention analysis explains the duration of each request by its transmission durations, the durations of the request-related operations on a server, and other computation performed by the server between the request arrival and the response to the client, which includes the server's overhead and computation on behalf of other *interfering* requests. With this knowledge, we can adopt alternate file layouts that reduce the contention among the requests (and presumably their duration).

We can use contention analysis to understand the performance of PIOUS programs. We first inspect the *Operation Map* produced by Carnival (see Figure 5a), where each line represents an operation that composes a class of requests. For each operation, the display presents its attributes (e.g.,

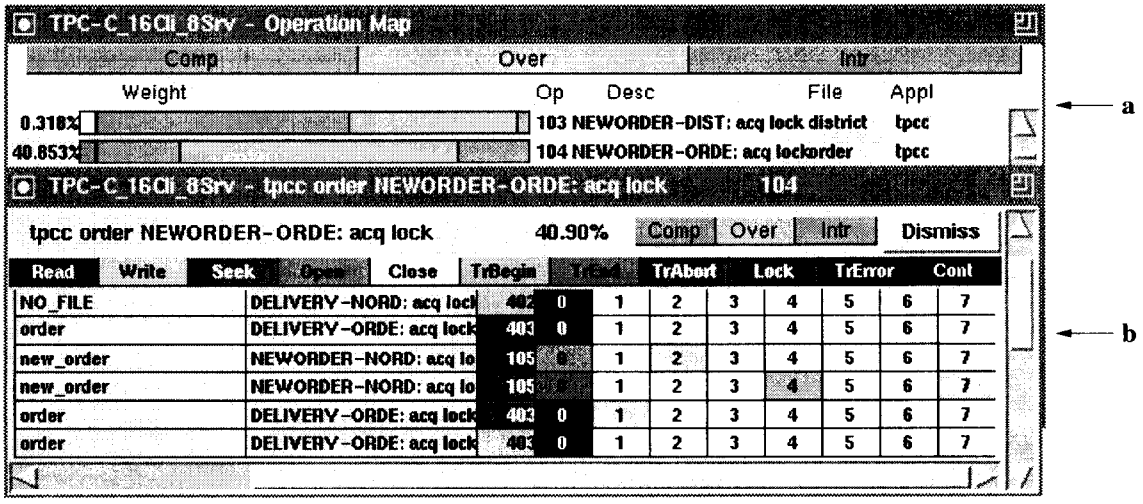

Figure 5: Contention Characterization from TPC-C

file, server), its cumulative duration, and a color bar that describes a breakdown of the cumulative duration into the three categories of interest (computation, system overhead, and interference). By browsing this display, we can easily determine the classes of requests that have the highest cumulative duration across the execution, as well as any significant interference, both subjects for further analysis. Clicking on the color bar produces summaries of the various characterizations for the operation displayed in the bottom of the window. A user can inspect a characterization by clicking on its summary, causing Carnival to pop-up a contention characterization (see Figure 5**b**), which lists the requests that interfere with this class. Information about the class of requests being explained is given at the top of the window, as is the relative weight of the interference.

Below this information, there is a color bar that serves as a reference for identifying the various PIOUS operations and the interfering classes of requests, one per line. Each line describes the attributes of the class (application, file, description, and basic block identifier), the percentage of interference caused by the class, and the amount of interference that occurred in each server. Each server is identified by number within the table; the intensity of the surrounding grey scale represents the percentage of the interference generated happened at that server (with darker squares corresponding to greater interference).

Next we describe how contention analysis helped improve the performance of TPC-C [8], a benchmark for on-line transaction processing systems. This benchmark is inspired by activities performed by a supplier of wholesale parts and simulates an environment where terminal operators execute transactions on a database, which include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock. There are five types of transactions in TPC-C: *new order*, *payment*, *delivery*, *order status*, and *stock level*. Each transaction occurs with a different frequency and accesses a different set of the following nine files: Warehouse, District, Item, Stock, Customer, History, Order, Order Line, and New Order. Four of these files, History, Order, Order Line, and New Order, are accessed in global mode, since transactions perform insertions on them during the execution.

In our implementation of TPC-C, each client represents a terminal operator that issues transactions continuously. We executed the TPC-C program on sixteen clients and eight servers, and each client performed twenty transactions. The experiments were performed on a cluster of eight DEC Alpha Server 2100 nodes connected by a DEC Memory Channel. Each Alpha Server node has four 233 MHz Alpha processors with 256 Mb of memory.

The operation map generated by Carnival (see Figure 5**a**) shows that 97% of the execution time is spent in accesses to Order, Order Line, and New Order. These files dominate the PIOUS operation costs because (a) they are the most frequent and (b) they are the most expensive (requiring access to global file pointers). Clicking on the operation map causes Carnival to display contention characterizations such as the display shown in Figure 5**b**. This figure shows a characterization for "acquire lock" operations from *new order* transactions, which account for 40.9% of the overall interference costs. This characterization indicates that this operation interferes with accesses to both Order and New Order. This interference happens more frequently on server 0 (indicated by the darker entries), which records the global pointer, but requests to all eight servers are affected. After verifying similar characterizations for other operations, we conclude that accesses to Order, Order Line, and New Order contend with accesses to all files, including themselves. We also find that accesses to District contend with accesses to Customer and vice-versa. Thus, in order to improve the performance of our TPC-C implementation, we have to isolate, as much as possible, Order, Order Line, and New Order from the other files. Furthermore, District and Customer should not share servers.

Knowing the sizes of files and which files and operations contend, we can determine the minimum number of servers that should be assigned to each file. For example, due to their size, Stock and Order Line require all eight servers. Based on both contention information and storage requirements, we determine file layouts that will allow better performance. In this case, since Order and New Order represent the majority of the accesses and usually contend with each other, we reserve a pair of servers for each of them, which are shared only with Stock and Order Line. The remaining files are spread across the other servers, each using at least two servers, while avoiding other minor sources of interference.

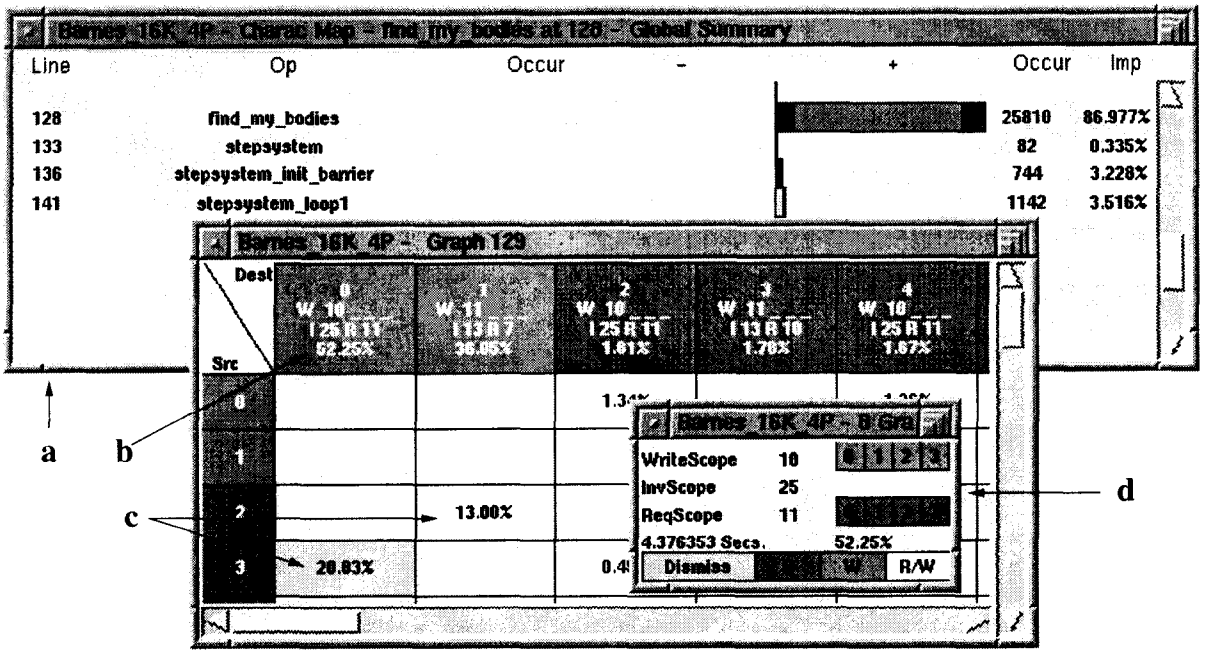

Figure 6: Synchronization and Communication Characterizations from Barnes

We executed the program using the new layout and the number of transactions per second (tps) increased for four of the five transaction types. Moreover, the total improvement (i.e., the weighted sum of the per-transaction improvements) is 40%, which reflects the overall improvement attained by the changes in the file layout. Only *delivery* exhibited worse performance, due to the reduction in the concurrency of accesses to New Order and Order. The contention characterizations for this last execution show that a lack of concurrency is the major source of the remaining contention, which is expected given that the number of clients is larger than the number of servers. Although there are other possible file layouts that satisfy the constraints inferred from the characterizations, they will not improve the performance of the program, since the main source of remaining contention is a consequence of the implementation of PIOUS.

4.2 Scheduling in Barnes

Our second example examines Barnes [19], an application that simulates the interaction of a system of bodies, in three dimensions, using the Barnes-Hut hierarchical N-body method. The computational domain of this application is modeled as an octree, where the leaves contain body information and internal nodes represent space cells. The experiments were performed on the same hardware described in Section 4.1. Applications are linked to an instrumented implementation of Treadmarks (version 0.9.6), which employs DEC's implementation of TCP/IP on the Memory Channel.

Execution of the original code for Barnes on four processors with 16384 bodies for 5-time steps takes 202 seconds, with an efficiency of only 39%. When we examine the execution profiles provided by Carnival, we find that processors are blocked on barriers for about 35% of the execution time, while another 25% of the execution time is spent communicating. Knowing that waiting time is a significant source of performance degradation, we proceed by analyzing the *Characterization Maps* (see Figure 6a). These maps provide color-coded operations for each code segment that is responsible for the observed waiting time: operations from the longer path (identified by "+") are on the right side of the window, and operations from the shorter path on the left ("-"). The number of occurrences of each operation is given, as is the percentage of the waiting time associated with each operation. From the analysis of these characterizations, we learn that two thirds of the waiting time is caused by differences in communication costs in the loops that traverse the octree, while the remaining third is caused by load imbalance in the same loops. Thus, communication is responsible, both directly or indirectly, for almost half of the execution time of the application.

Synchronization analysis also identifies a single function, `find_my_bodies`, as a main source of communication overhead, being responsible for 45% of the communication costs (see Figure 6a). Profiles also show that the variable `bodytab`, an array where all bodies are stored, accounts for 75% of the overall communication costs.

At this point, we know that communication is a major source of overhead and the variable `bodytab` is responsible for most of it. We inspect the communication characterizations of `bodytab` (Figure 6b,c, and d). A communication characterization, which is organized as a table, presents information about a variable (or set of pages). Each graph is represented as an incidence matrix, where the column header (Figure 6b) identifies the access pattern (using a color code), the source code location of the faults (R for request, I for invalidation, and W for preceding writes), and the percentage of total page fault cost in the graph associated with that vertex. The entries (Figure 6c) quantify the relative frequency of transitions between vertices in the graph. It is also possible to obtain per-processor information by clicking on the header of each column (Figure 6d). After analyzing the access patterns to `bodytab`, we found that most of the accesses in `find_my_bodies` have a multiple-producer-multiple-consumer pattern (as seen in figure 6b), and all (or almost all) processors read or write to each page of `bodytab` (as seen

in figure 6d). Since each page contains several bodies, false sharing occurs as a consequence of the dynamic distribution of bodies to processors, causing several processors to access each page during every iteration of the algorithm. It is not surprising that a program written for a shared-memory machine with small coherence units presents false sharing when executed on a machine with large coherence units. However, in porting programs such as Barnes, which may consist of several thousand lines of code and many shared variables, it can be difficult for programmers to determine the code segments and variables that need attention without some automated support.

One approach to minimizing this false sharing is to avoid changing the allocation of bodies to processors at every iteration and to consider page boundaries when distributing the bodies among processors. Although the static distribution of bodies increases the load imbalance among processors, we expect this increase to be compensated for by a reduction in communication costs, which dominate the execution time. Following up on this analysis, we added a 25-line function that distributes bodies statically, and disabled the per-iteration dynamic redistribution. The modified program executed 28% faster than the original version. As shown by the Carnival profiles, we observed that this improvement comes from the reduction of the waiting times at barriers by more than one third, and the reduction of communication costs by about a half. This example illustrates how synchronization analysis and communication analysis can be used to find the sources of excess communication in the source code and suggest changes. In this particular example, CEA lead us to focus on a single small procedure, even though profiles would have suggested a focus on barriers located elsewhere in the program.

4.3 Tuning Squid Hierarchies

Squid [6] caches Internet data by acting as a proxy server for Internet users. That is, whenever a user requests a page to a WWW browser, the browser asks Squid to get the page. If the requested page is available at the proxy server, Squid returns it. Otherwise, Squid connects to the remote server and requests the page. It then transparently streams the data through itself to the client machine, while keeping a copy of the data. The next time a request is made for that page, Squid simply reads it off disk, transferring the data to the client machine almost immediately.

Squid servers can be arranged hierarchically for enhancing response time and reducing network traffic. The hierarchy works as one cluster that answers requests, and pages can be cached at one or more levels of the hierarchy. Also, each machine in the hierarchy can be assigned to a specific range of domains (e.g., `.com`, `.net`) so that it is possible to balance the load and exploit the reference locality inherent to accesses. Thus, there are two configuration parameters for each machine in a hierarchy: the relationships that each machine has with other machines (sibling, parent) and the domain(s) that are served by each machine. Note that there is no standard recipe for configuring a Squid hierarchy, because machine configurations, network resources, and traffic characteristics may vary drastically among sites. Our experience is that system managers usually tune hierarchy parameters using trial-and-error, which is both laborious and error-prone.

We used CEA techniques to identify sources of performance degradation in Squid hierarchies and to determine configuration parameters that allow an increase in the quality of service provided by those hierarchies. In particular, our goal was to find the best configuration for a cluster of four Pentium machines connected by a 100MB fast Ethernet. Each of these machines has 300Mb of disk for WWW caching and is also connected to an Ethernet network, from which requests arrive. All machines run FreeBSD 2.5.2 and our instrumented Squid 1.1.17. Both clients and WWW servers are emulated by other workstations in the same LAN.

The workload used in our tests is based on logs from POP-MG [1], which is the Internet backbone that serves the state of Minas Gerais, Brazil. POP-MG has several national and international links that represent a bandwidth of up to 9 Mbps and an average traffic rate that is close to 6 Mbps. The average load of the caching proxy servers is 1,800,000 requests per day. We analyzed a log containing 4,235,511 requests for 1,079,044 unique objects, which results in about 12 Gbytes of data that has to be stored in our machines. Since our machines can only store up to 1.2 Gbytes, the cache holds up to 10% of all data. For our experiments, we characterized the workload, and generated a smaller stream of requests that is feasible for experimentation, while maintaining the same cache ratio. In the case of POP-MG for example, we generated a set of 96,000 requests that follow the workload and determined the total size of unique objects ($\approx$ 400 Mbytes). We then used 10% of this size as our total cache size, leading to 10 Mbytes of disk cache per machine.

Since we did not know in advance how to configure a hierarchy to obtain the best performance, we started with a flat configuration of four children. This configuration was able to answer a request in 2.8 seconds on average. We start our analysis by verifying the *Request Map* produced by Carnival, which displays both static attributes (e.g., domain, source host) and profile data (e.g., number of requests, cumulative response time) associated with the various classes of requests. There is a color bar that represents a breakdown of the cumulative response time into five categories: computation, synchronization, contention, communication, and server overhead. By clicking on this display, the user can verify both contention characterizations, which show classes of requests that contend in the servers of the hierarchy, and waiting time characterizations, which explain the causes of delays in queries between servers.

Both synchronization and contention analysis of the most frequent classes of requests point to sibling requests as a main source of performance degradation, representing 33% of the overall response time. Knowing that sibling requests represent a source of performance degradation, we examine the communication graphs produced by communication analysis, which tell the user the access patterns of the objects stored in the hierarchy. Figure 7 presents the communication graph for a set of objects frequently requested. We can observe that most of the time (73%) the objects are either available locally (1–Cached Locally) or replicated in the other sibling caches, (4–Cached Remotely(3)) and the graph transitions are usually among these two states (49%). In summary, there is too much replication in the hierarchy, and a large number (50%) of the cached pages are replicated in more than one server.

Since the most frequently accessed web pages tend to be the most popular web pages (and therefore are accessed widely), the flat hierarchy ends up storing the same documents in all of the siblings. One strategy to minimize the excessive replication of pages is to configure the machines as a hierarchy, where pages accessed less frequently are stored in the upper levels of the hierarchy.

We then configured our four machines as a two-level hi-

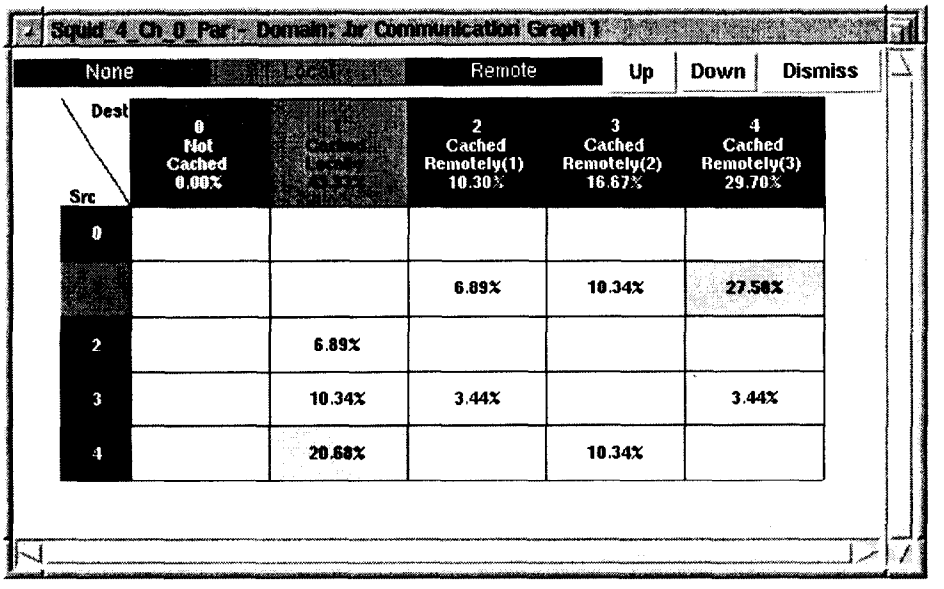

Figure 7: Communication Characterizations for Squid Cache Proxy

erarchy with three children and one parent. Surprisingly, the average response time of this configuration increased to 4.1 seconds. The profiles provided by Carnival indicated contention as the major source of performance degradation and our analysis of the contention characterizations confirmed that simultaneous accesses to the single parent were the main source of contention, accounting for 36% of the response time on average. Moreover, the same characterizations also showed that sibling-related interference was responsible for 20% of the response time. Also, the communication graphs showed that the utilization of a hierarchy reduced the amount of replication by only 10%, resulting in frequent requests to the parent.

In order to reduce contention at the parent, we configured a hierarchy with two parents and two children, where each child communicates with its sibling and the parents. The average response time decreased to 2.3 seconds, a 22% improvement over the initial configuration, although each child had to handle twice the number of requests. In this case, contention among requests increased, accounting for 5% of the response time (compared to 1% in the flat configuration), while both sibling intrusion and contention for parents decreased (7% and 23%, respectively).

In this example, CEA helped us to understand the dynamic behavior of a cache hierarchy. Since the performance of the hierarchy depends on the client workload, the observed phenomena and their causes may vary significantly across sites, making our automated approach extremely valuable for these types of analyses.

5 Related Work

Much of the work intended to assist the programmer in understanding the relationships between execution events in parallel and distributed programs is based on *causality*, as defined by Lamport's *happened-before* relation [9]. Parallel debugging tools and techniques [10, 15] often employ causality graphs to delimit the sets of events that must be inspected in order to find the source of execution errors or to detect potential race conditions. Critical path analysis [20] uses causality graphs to identify the execution path that dominates program performance, focusing attention on the code segments where tuning should be performed.

There are other tools and techniques that express dynamic relationships among program operations and data. The PPUTTS toolkit [11] provides a graphical view of the program behavior based on a directed acyclic graph representation of processes and communication operations. Another example is StormWatch [3], which is a visualization tool for memory system protocols that presents multiple views of memory access operations, including performance slices that capture relationships between individual memory events, exposing causality in memory operations. The main limitation of these two tools, and other event-based tools, is that they can overwhelm the programmer with detailed data.

Rajamony and Cox [18] implemented a performance debugger that automatically identifies code transformations that reduce synchronization and communication. This performance debugger is similar in approach to our work, although it is specific to the target programming environment and focuses on one class of overheads.

Our work is distinguished primarily by its emphasis on understanding the durations of events, rather than the occurrence of events. Our techniques are designed to assist the programmer in improving program performance by reducing the durations of events that cannot (in general) be eliminated entirely. The novelty in our approach is to use causality information, augmented by appropriate timing information, to reason about the underlying causes of performance phenomena.

6 Conclusions

In this paper we presented a novel approach for performance understanding called cause-effect analysis. We illustrated our approach by analyzing the performance of a parallel file system, a DSM parallel application, and a WWW cache proxy server. Our experience demonstrates that these techniques can be used effectively to understand the causes of

poor performance, and to identify specific improvements in the source code. We are currently working on the use of CEA techniques in other environments, and are also investigating the utilization of characterizations for performance prediction.

References

[1] V. Almeida, M. Cesário, R.Fonseca, W. Meira Jr., and C. Murta. The influence of geographical and cultural issues on the cache proxy server workload. *Proc. of WWW7*, 1998.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *IEEE Computer*, February 1996.

[3] T. Chilimbi, T. Ball, S. Eick, and J. Larus. Stormwatch: A tool for visualizing memory system protocols. *Proc. Supercomputing'95*, San Diego, CA, December 1995.

[4] M. Crovella and T. LeBlanc. Performance debugging using parallel performance predicates. *Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140–150, May 1993.

[5] M. Crovella, T. LeBlanc, and W. Meira Jr. Parallel performance prediction using the Lost Cycles Toolkit. TR 580, Department of Computer Science, University of Rochester, May 1995.

[6] National Laboratory for Applied Network Research. Squid Internet Object Cache. Information available at http://squid.nlanr.net/Squid/.

[7] S. Graham, P. Kessler, and M. McKusick. gprof: a call graph execution profiler. *Proc. SIGPLAN '82 Symp. on Compiler Construction*, pages 120–126, June 1982.

[8] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann, second edition, 1993.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, July 1978.

[10] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[11] T. LeBlanc, J. Mellor-Crummey, and R. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, June 1990.

[12] W. Meira Jr. *Understanding Parallel Program Performance Using Cause-Effect Analysis.* TR 663 (PhD thesis), Dept. of Computer Science – University of Rochester, Rochester, NY, July 1997.

[13] W. Meira Jr., T. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in Carnival. *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools*, pages 1–10, May 1996.

[14] W. Meira Jr., T. J. LeBlanc, N. Hardavellas, and C. Amorim. Understanding the performance of DSM applications. *Proc. IEEE Workshop on Communication and Architectural Support for Network-based Computing (CANPC)*, volume 1199 of *Lecture Notes in Computer Science*, pages 198–211, February 1997. Springer-Verlag.

[15] B. Miller and J. Choi. A mechanism for efficient debugging of parallel programs. *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 141–150, May 1988.

[16] S. Moyer and V. Sunderan. PIOUS for PVM: User's guide and reference manual – Version 1, 1995. Technical report, Emory University, 1995.

[17] John K. Ousterhout. *Tcl and Tk Toolkit.* Addison Wesley, 1994.

[18] R. Rajamony and A. Cox. Performance debugging shared memory parallel programs using run-time dependence analysis. *Proc. 1997 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, June 1997.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *Proc. 22nd Annual Int'l Symp. on Computer Architecture*, pages 24–36, June 1995.

[20] C. Yang and B. Miller. Critical path analysis for the execution of parallel and distributed programs. *Proc. 8th Int'l Conf. on Distributed Computing Systems*, pages 366–373, June 1988.