Reproducing inter-process synchronization for performance prediction using lightweight system call tracing

Ariel N Burton Paul H J Kelly

Department of Computing, Imperial College, London, UK
{anb,phjk}@doc.ic.ac.uk

Abstract

This paper provides a brief overview of techniques and tools being developed for monitoring and predicting the performance of UNIX server configurations for given real-life workloads. We show how our system call trace mechanism, called ULTra, captures a complete trace of a process's calls to the operating system with only minimal interference to the system under study. Once captured, the traces can be used to reproduce the captured workload's behaviour in full.

Rerunning such multi-process workloads from their traces is complicated because the inter-dependencies between the activities of the individual constituent processes must be reproduced correctly if the overall behaviour of the workload is to be reproduced successfully. We show how our approach can be extended to meet this requirement, and allow multi-process workloads to be traced and rerun. To illustrate the usefulness of our tools, we present a case study in which our traces are used to predict the impact of file system caching on a multi-process WWW server's performance.

1 Introduction

Our aim in this work is to develop a tool for a system performance consultant to use to characterize workloads that are complex and subject to external influences and stimuli which cannot be controlled, managed, or predicted easily. The consultant would install the tool, and would monitor the system as it performs its normal duties. The consultant would use the information captured by the tool to evaluate the effectiveness of changes such as hardware upgrades, adjustments to the system's configuration or tuning parameters, or workload redistribution to improve performance.

The evaluation methodology presented in this paper characterizes a workload by the trace of its system calls. By rerunning the sequence of system calls in a trace under different conditions, it becomes possible to study the performance of the workload under different system configurations. We distinguish two modes of rerunning traces: trace *replay* and *reexecution*. These are described below.

Trace replay In this simple rerun mode, each call in the trace is reissued in turn, and user-level inter-system call execution time is simulated by simply looping for the appropriate period as recorded in the trace. The actual time taken to complete trace replay depends on the system call service times achieved by the system under test.

Trace reexecution In some applications, spinning to account for user-mode execution leads to inaccurate results because the application interacts with the operating system in other, less explicit ways, for example, by causing TLB misses or page faults, or by flushing operating system data from hardware caches. We reproduce this behaviour by reexecuting the original application code.

In order to get reproducible results, we make sure that results returned from system calls are recorded in the trace. The reexecuting application should behave in a precisely reproducible way since it is fed precisely the same inputs.

The trace needed here is simpler since user-level execution times are not required. System call parameters need not be recorded since they will be supplied by the reexecuting application. Results, however, must usually be recorded to ensure that the application receives the same inputs.

Although most applications of interest can, at least in principle, be reexecuted efficiently, some behaviours are problematic and we return to this question when we discuss future developments in Section 6.1.

Multiprocess workloads Our earlier work [8, 9] focused on single-process and sequential multi-process workloads. Here we examine workloads in which several traced processes may be running concurrently and interacting with one another. Each process logs its system calls to a different file, so the trace is only partially ordered. However, we have to arrange synchronisation between reexecuting processes for two reasons:

1. where one process waited for another at trace capture time, we should reproduce this dependency at rerun so that only feasible execution orders are exercised.

2. to keep the trace file size and trace capture overheads small, we avoid logging data read from files whenever possible, relying instead on reexecuting the read. For this to work, we have synchronise to ensure that the correct data are used.

1.1 Contributions of this paper

The focus of this paper is the question of how to trace and rerun (both replay and reexecute) multi-process applications:

1. We identify the constraints and problems in replaying or reexecuting concurrent traces, and demonstrate that timestamping system call entry and exit is not adequate

2. We present an off-line algorithm for finding synchronisation dependences between traces, using semantic knowledge of the traced operations

3. We describe how modest changes to the OS kernel were systematically applied to capture the information necessary to determine precedence between dependent operations

4. We present our experience in developing an efficient technique for reproducing the partial process dependence order at rerun time

Finally, we demonstrate the effectiveness of the performance evaluation tool using a multi-process `WWW` server running with varying amounts of RAM for file caching, and evaluate the predictive value of the technique.

2 Related work

Trace capture has been used for many years for performance evaluation. The critical aspect of our work lies in capturing just enough information, in this case system calls, to be able to reconstruct the complete computation by reexecution. Rather than supplanting lower-level trace capture and analysis, for example by hardware monitoring or modifying microcode, this facilitates it by making a reproducible record of the original workload. We therefore focus our literature review on trace capture and reexecution.

Intercepting system calls The `ptrace()` system call provides a mechanism for one process to monitor the system call activity of another, but incurs large overheads [8]. Jones [11, 12] describes a general technique for interposing agents between an application and the operating system using a generic mechanism to redirect calls to a specified handler. Ashton and Penny [1] developed INMON, an "interaction network monitor", designed to trace the activity in the kernel caused by individual user actions. Tools of this nature complement our work in that they provide an insight to activity within the kernel caused by a workload, whereas we report trace capture in order to characterize the workload.

File access trace studies Ousterhout et al. [13], Baker et al. [3] and Bozman et al. [7] used traces in file system performance analysis. Of more interest is DFSTrace, used by Mummert and Satyanarayanan [15] in the evaluation of the Coda file system, since they also replayed the traces using the timing information given by the trace. Instead of modifying the operating system kernel, Tourigny [17] and Blaze [6] exploited a remote file system architecture to obtain traces of file system activity by monitoring the interactions between clients and server.

By contrast, we aim in this paper to capture the entire system call trace, and to use it to study the overall system performance by using it to reexecute the application.

Logging reexecution for fault-tolerance Logging for reexecution or rollback has long been used for recovery from faults, and is common in transaction processing systems. Closer to our work are attempts to do this *via* a standard UNIX-like API; an interesting example is the QuickSilver system [16]. When concurrent processes are involved, techniques from checkpointing in distributed systems (*e.g.*, see Johnson and Zwaenepoel [10]) will also be relevant.

Replay for debugging The problem of reexecution of parallel UNIX processes is similar to that of replaying parallel programs (*e.g.*, see LeBlanc and Mellor-Crummey [14]) for debugging purposes. Note, though, that we need to be able to reproduce the original execution time as accurately as possible.

Finally, Bitar [5] gives a useful review of the validity issues in trace-driven simulation of concurrent systems.

3 ULTra

For our approach to be viable and attractive, the tool must incur minimum risk and interference to the system under examination, provide enough information for the performance tuning mechanisms to be exercised properly, and lead to results having adequate predictive accuracy

Trace capture ULTra (User Level Tracing) intercepts system calls and writes trace information to a trace file. Its performance depends on two key factors:

1. an efficient mechanism for intercepting the workload's system calls,

2. a buffering scheme to reduce the number of `write` operations required to record the trace.

It is the second factor which complicates rerunning multi-process because each process has its own trace file, and therefore the trace is only partially ordered.

To be easy to use, we need a simple mechanism for controlling tracing. Having considered various alternatives, we chose to substitute the dynamically-linked standard shared library providing UNIX system calls. In the ULTra version the system call stubs are extended with modifications for trace capture and reexecution.

3.1 Trace reexecution

In order to reproduce both the workload's explicit and implicit interactions with the operating system, the original application's code is reexecuted. In order for this to work, the application's environment must be recreated from the traces. System calls are reissued but the values returned to the application are taken from the trace. Some system calls, however, will return different values because, for example, the call returns a kernel-created handle for some resource (*e.g.*, `fork()`). In general, there is no way of ensuring that when the call is reissued, the kernel selects the same value. Calls of this type are handled by keeping a translation table mapping trace capture values to trace reexecution values.

Handling synchronization A more important problem is that any inter-process synchronization at trace capture time must be honoured. This synchronization can be either explicit, or implicit:

Explicit this occurs when, at trace capture time, one process waited for another.

Implicit this occurs when one process read data (*e.g.*, file data or metadata) which were modified by another process. The processes may not have synchronized explicitly, and the effect we are trying to reproduce is the outcome of a race.

The key issue is that for reexecution to succeed, we must ensure that these synchronizations are reproduced and the processes enter each critical region in the same order so that the overall behaviour of the workload is preserved. This can overconstrain the order of events during reexecution, since as far as an application is concerned it does not always matter in which order the events occur, for example, when writing records to a log file.

Identifying inter-dependencies We post-process the traces to identify any dependencies between the processes in the workload. In general, the order of actions performed by two processes must be reproduced if they both refer to the same object, and one of them modifies the object. To do this we modified the operating system kernel to timestamp each operation on the underlying resources. This instrumentation must be positioned carefully in the kernel for two reasons:

1. if the timestamps were recorded at user-level the process could be descheduled between this point and when the operation is initiated. In the intervening period another process could access the resource.

2. the order in which the operations on a resource are started is not necessarily the same as that in which they terminate. This is because operations may overlap in the kernel, and the order in which the requests are processed may depend on other factors, *e.g.*, the current position of a disk arm. The instrumentation to acquire the timestamps must be positioned at the point(s) were the operation is committed.

The modifications are very straightforward, few in number, and can be applied systematically (see Section 4).

Resource granularity We control the degree we allow the global ordering of events during rerun to diverge from that at trace capture time by varying the granularity of what we consider an object. For example, if we are interested in reproducing the global order of events, we would consider the entire operating system kernel as a single resource. It is more useful, however, to relax the ordering during rerun as much as possible so as to allow the workload to execute as naturally as possible on the new configuration. Providing the order of events on each resource is preserved, the global ordering of the events during rerun can be relaxed.

For our purposes it is sufficient to consider just the files (inodes) as the resources in the system. We annotated the trace records for each operation with its timestamp. Figure 3.1 shows an example in which a single resource is used by five processes. The figure shows the user file descriptor tables and file table entries that would be constructed by a conventional UNIX kernel [2]. Processes P_1 and P_2 are descended from a common ancestor that opened the file for reading, and each accesses the file using an inherited file descriptor. Similarly for processes P_4 and P_5. Process P_3 opened the file independently for writing. Events on other resources are not shown since these can proceed independently.

In this example there are a number of different handles, or file descriptors, associated with the file. Operations using the same file descriptor must be sequenced correctly even in the case of reads because the file pointer is advanced as a side effect. Reads using distinct file descriptors should be allowed to execute in any order, providing their ordering relative to the writes is preserved. However, since the traces record only the sequence of operations on the underlying file, the events must be rerun in the order shown. We term this *coarse* rerun.

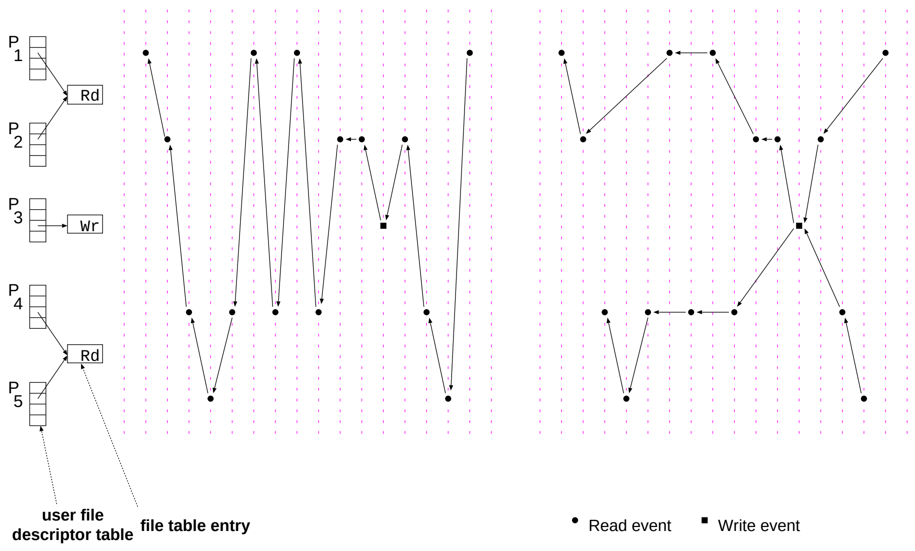


Figure 1: Coarse vs fine dependence analysis. The left-hand graph shows the coarsely ordered dependencies for a single resource, the right-hand graph shows the same sequence of events ordered using the fine dependence analysis.

We can relax the constraints imposed by coarse ordering by post-processing the traces. In this procedure the traces are fed through an analyzer which simulates the effects of the file system related operations on the kernel's tables. This allows us to identify reads which use common file descriptors, and to arrange for these operations to be sequenced correctly. Reads using other file descriptors are allowed to proceed independently. Figure 3.1 shows on the right hand side the same set of operations, but with the revised dependencies after post-processing the traces. It can be seen that under this ordering the workload is allowed greater freedom to execute naturally in that reads by processes P_1 and P_2 are allowed to proceed independently of the reads by processes P_4 and P_5. Note, however, that the pair of processes P_1 and P_2 (and also P_4 and P_5) must coordinate their reads because they share the same file pointer. What we have done is to use semantic knowledge of the UNIX kernel to identify which reads are truly independent. Potentially, we could refine the post-processor to identify reads and writes which refer to distinct regions of the file. This would allow us to relax the relative ordering of these operations. We term this *fine* rerun.

3.2 Trace replay

In this form of trace rerun, we simply reissue the system calls made by the original workload, and we simulate user-level execution time by spinning on a loop for the appropriate period. We use the same techniques as trace reexecution to handle inter-process synchronization. Trace replay exercises the operating system as before, but as the user-level application code is not executed we cannot reproduce behaviour which depends on processes' memory access patterns. Thus, trace replay is potentially less accurate than trace reexecution because we cannot reproduce paging, caching, TLB, *etc.* effects. The extent to which this is significant depends on the characteristics of the workload.

Measuring time - accounting for pre-emption For trace replay to be accurate we must ensure that the system calls are reissued at the correct rate. This happens naturally for trace reexecution, but for trace replay we need accurate, high resolution measurements of the processes' user-level inter-system call execution times. This valuable information is not provided in standard Unix implementations (user time is measured by sampling every few milliseconds).

We account for user time in the presence of other processes by modifying the kernel to update a timer in its process table entry on each context switch to, or from, user mode. To keep the overhead to a minimum, the cost of reading the clock should be low. We describe how this is achieved in our implementation in section 4. This provides accounting for user-mode execution time at clock-cycle resolution. The counter could be accessed via a system call, but we improve performance by avoiding this. Instead, immediately prior to returning from a system call the kernel writes the times to the `ultra_area`, a small, pre-determined area of the process's user level address space reserved for this purpose. When the system call returns, these times can be read from the region by ULTra, and recorded in the trace. It should be noted that if the application is not being traced, then the times are simply ignored. The location of this region is carefully chosen (for example, at the base of the stack) so that its presence is transparent to both traced and untraced applications.

4 Implementing ULTra

ULTra is currently implemented as two components: a substitute for the `libc` (version 5.3.12) shared library running under LINUX version 2.0.35, and a small number of kernel modifications. In addition we have developed a suite of tools for analyzing our traces.

Kernel modifications The LINUX system call mechanism was modified to include the time measurement extensions described in Section 3.2. To measure time with high resolution and low overheads, we exploit the PENTIUM processor's 64 bit Time Stamp counter. This is incremented on every clock cycle, and can be read in a single instruction (`rdtsc`). This allows us to obtain fine-grained times very efficiently. We use this feature to determine the number of clock cycles a process spends executing at user level.

We also instrumented the kernel to generate timestamps for the resources used by the workload. We identified the critical regions within the kernel where it was important that we record the order in which events occurred to be those that involved operations on inodes. About 60 points were identified and instrumentation was inserted to generate and assign a timestamp each time these operations were performed. We could have used a simple counter, but instead we used the value reported by `rdtsc`.

The user-level execution times and resource acquisition timestamps were communicated to the user-level component of ULTra through the reserved `ultra_area` described earlier. In all, the modifications were modest, amounting to about 300 lines of `C` and PENTIUM assembler.

The library This component is responsible for marshalling and writing the trace records. In a naïve implementation, the trace records would be written out as soon immediately. Doing so would double the number of real system calls made by the workload, leading to poor performance. Consequently buffering is used to reduce this overhead. Surprisingly, buffering is ULTra's main source of complexity. The areas most affected are process creation (`(v)fork()`) and program invocation (`execve()`) where the buffer must be handled carefully to protect it from corruption. Trace capture, reexecution, and replay are all affected, but there is insufficient space to explain the details here.

4.1 Implementing trace rerun

An important consequence of our decision to use buffering to improve performance is that each process in the workload has its own trace, and therefore the traces are only partially ordered. We post-process the traces to identify the dependencies between the processes. The traces are modified so that records for system calls which use shared resources are augmented with the identity of the operations on which they depend. The rerunning processes synchronize their accesses to the resources using a table in shared memory. Entries are posted in the table when a process completes an operation. A process about to attempt an operation examines the table to determine whether the events on which it depends have completed. If so, then it initiates the operation, otherwise it waits (by yielding the CPU) until the events in question have been posted.

5 Using ULTra to predict performance

Choice of benchmark ULTra is designed for workload characterisation in situations where the application is interacting with its environment in complicated ways which make it difficult to redo performance experiments with precisely reproducible results. However, for the purposes of this paper, we need to be able to compare the execution time of a particular workload with the execution time using replay or reexecution of an ULTra trace. Thus we need to be able to reproduce the actual workload as well.

We chose the `apache` web server as the benchmark in order to overcome this problem; it has the advantage that we can rerun it with a repeated sequence of HTTP "GET" requests, and get exactly the same behaviour (a simple illustrative example of a situation where this would not work would be where `apache` is configured to operate as a `WWW` proxy cache; it is difficult to get precisely reproducible results because cached data expires as time elapses).

Two workloads We configured `apache` (version 1.2b6) to manage a copy of the 11,110 managed by the `WWW` server of the Advanced Languages and Architectures (ALA) section of the Department of Computing at Imperial College. This amounted to approximately 175MB. The `apache` server was configured to run in multi-process mode, with five processes to handle the HTTP requests from the clients. In order to conduct repeatable experiemnts, a set of simple clients running on the same CPU were used to issue a sequence of 5,000 requests derived from the access logs of the ALA server. Deriving the requests in this way ensured that the patterns of access to the documents were realistic.

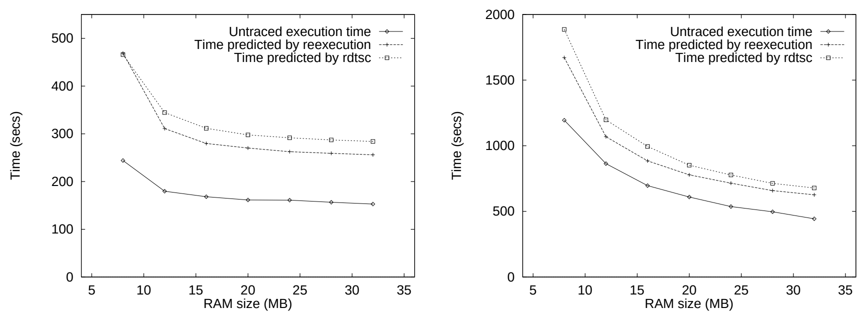
In order to illustrate a richer range of behaviours, a further workload for `apache` was used. This workload was designed to have higher demands on memory (see 'Configuration modification' below). In this variant, the server was configured to manage about 4,900 documents, amounting to approximately 32MB. A list of queries was constructed so that each document was accessed once. Different random permutations of this list was used by each of the clients. As before, `apache` was configured as five processes.

Configuration modification `apache` is highly file intensive, and there is potential for caching since certain URLs are requested repeatedly during the experiment. `apache` relies on the underlying file system to cache repeatedly-used files, and this depends on having enough memory. As an illustration of the potential value of the approach, we show here that the ULTra trace can be used to predict the performance of the workload on configurations with a range of RAM sizes. We booted LINUX with various amounts of RAM, and compared the execution time of the actual workload with the time taken to replay the ULTra trace, and to reexecute it. The same rerun trace was used for each memory size, captured from a run with the minimum 8MB configuration. Coarse ordering was used when rerunning the traces.

5.1 Results

The experiments reported here were performed on an unloaded IBM-compatible PC with a 166 MHz Intel PENTIUM CPU, 32MB of EDO RAM, and a 512KB pipeline burst-mode secondary cache, running LINUX version 2.0.25 (or variants thereof). All application file input and output was to a local disk, with ULTra traffic directed to a second, local disk. Elapsed times were measured using a statically linked instance of version 1.7 of the GNU standard UNIX timing utility, `/usr/bin/time`. `apache` was built using the default configuration and `make` options, though a small modification was made to the source to ensure that termination could be handled conveniently.

Figure 2 shows the actual and predicted time achieved by `apache` for the two workloads. It can be seen that both trace reexecution and trace replay are successfully predicting the effect varying the availability of RAM has on `apache`'s performance. The accuracy of the artificial workload is considerably better than that for the ALA workload. The reason for this difference lies in how the two workloads are affected by the file cache. It can be seen that the working set of the ALA workload can be accommodated in memory for RAM sizes greater than or equal to 20MB. This is being identified correctly by ULTra. Under LINUX, file system operations that can be satisified from the cache do not block, and usually return directly to the calling application. This affects the



(a) ALA workload (b) Artificial workload

Figure 2: `apache` performance with varying RAM—predicted and actual

ALA workload because processes which have yielded are subject to following effects caused by the yield mechanism used to order the events during rerun:

1. When a process yields, its dynamic priority is cleared. The LINUX scheduler does not recalculate the process's priority until the time-slices of all runnable proceeses in the systen have expired [4]. One consequence of this is that a process may be forced to wait until long after the event for which it is waiting has occurred. This, in turn, can affect the other processes in the workload, which may depend on events to be performed by this process.

2. In this scheme, there can be only one operation pending on a resource at any one time since an event is not started until the previous ones have completed fully. This excludes the possibility of overlapping operations in the kernel.

In particular, recalculation of the processes' priorities, and hence their opportunity to run, is delayed. In contrast, the artificial workload has a very much larger working set, and therefore file system operations block more frequently. This allows the yielded processes to execute more frequently, thereby reducing the effects of delays introduced by the rerun mechanism. Additionally, since there is less locality in the artificial workload the loss of opportunity to overlap operations in the kernel is less significant. `apache` is particularly affected by these effects because the processes synchronize frequently using a lock file to coordinate their use of a shared network socket from which the HTTP requests are read.

6 Conclusions

We have presented the design of ULTra an efficient, portable technique for capturing traces of system call activity of multi-process UNIX workloads. ULTra's efficiency is achieved by running at user level as part of the standard libraries linked to applications, and also by buffering the output of trace information. We describe how we can determine the inter-process dependencies by post-processing the traces, and instrumenting a small number of kernel critical regions.

An important area where ULTra may be applied usefully is in the performance evaluation, tuning and comparison of operating systems and file systems. We present a case study illustrating this, and demonstrate that ULTra can be used to capture the workload without substantial interference, and can be used to give fairly accurate predictions of the effect of configuration changes on application throughput.

6.1 Further work

Paging activity Trace replay is potentially inaccurate compared with reexecution because it does not capture paging behaviour. We are working on introducing additional instrumentation to track a process's memory access behaviour. Preliminary results are very promising.

Asynchronous signals Workload-determined signals, such as timer interrupts, are problematic since there is potential for inconsistent results when the trace is replayed on a faster or slower system. Implementation-determined signals, such as synchronisation between processes, are easily traced. For reexecution, it is vital for the signal to be delivered at precisely the same instruction execution point as during trace capture. Pre-emptively scheduled threads can be handled by a similar mechanism.

Given that it is difficult or impossible to create a reexecutable trace for absolutely any application, our aim is to be able to detect whether an application behaves in a way which invalidates the trace.

References

[1] P. Ashton. The Amoeba interaction network monitor—initial results. Technical Report TR-COSC 09/95, Deptartment of Computer Science, Univ. of Canterbury, New Zealand, Oct 1995.

[2] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. 13th ACM Symposium on Operating System Principles*, pages 198–212, Oct 1991.

[4] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *LINUX Kernel Internals.* Addison-Wesley, second edition edition, 1998. Translated from the German.

[5] P. Bitar and A. M. Despain. Multiprocessor cache synchronisation; issues, innovations, evolution. *Computer Architecture News*, 14(2), June 1986. 13th Annual International Symposium on Computer Architectures.

[6] M. Blaze. NFS tracing by passive network monitoring. In *USENIX Winter Conference*, pages 333–334, 1992.

[7] G. Bozman, H. Ghannad, and E. Weinberger. A trace-driven study of CMS file references. *IBM Journal of Research and Development*, 35(5/6):815–828, Sept/Nov 1991.

[8] A. N. Burton and P. H. J. Kelly. Workload characterization using lightweight system call tracing and reexecution. In *IEEE International Performance, Computing and Communications Conference*, pages 260–266. IEEE, February 1998.

[9] A. N. Burton and P. H. J. Kelly. Tracing and reexecuting operating system calls for reproducible performance experiments. *Journal of Computers and Electrical Engineering—Special Issue on Performance Evaluation of High Performance Computing and Computers*, 1999. To appear.

[10] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *J. of Algorithms*, (11), 1990.

[11] M. B. Jones. *Transparently Interposing User Code at the System Interface.* PhD thesis, School of Computer Science, Carnegie Mellon University, Sept 1992.

[12] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. *Proc. 14th ACM Symposium on Operating System Principles*, 27(5):80–93, Dec 1993.

[13] J. K.Ousterhout, H. D. Costa, D. Harrison, J. A. Knuze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. In *Proc. 10th ACM Symposium on Operating System Principles*, pages 15–24, Dec 1985.

[14] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. on Computers*, C-36(4):471–482, Apr. 1987.

[15] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software—Practice and Experience*, 26(8):705–736, June 1996.

[16] F. Schmuck and J. Wyllie. Experience with transactions in QuickSilver. In *Proc. 13th ACM Symposium on Operating System Principles*, pages 239–53, Oct. 1991.

[17] S. R. Tourigny. Characterising the workload of a distributed file server. Master's thesis, Deptartment Computational Science, Uni. of Saskatchewan, Canada, Sept 1988.