WORKLOAD CHARACTERIZATION USING LIGHTWEIGHT SYSTEM CALL TRACING AND REEXECUTION

Ariel N. Burton and Paul H. J. Kelly
Department of Computing, Imperial College, London, United Kingdom
{anb,phjk}@doc.ic.ac.uk

ABSTRACT

This paper shows how system call traces can be obtained with minimal interference to the system being characterized, and used as realistic, repeatable workloads for experiments to evaluate operating system and file system designs and configuration alternatives.

Our system call trace mechanism, called ULTra, *captures a complete trace of each* UNIX *process's calls to the operating system. The performance impact is normally small, and it runs in user mode without special privileges.*

We show how the resulting traces can be used to drive full, repeatable reexecution of the captured behaviour, and present a case study which shows the usefulness and accuracy of the tool for predicting the impact of file system caching on a 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 a customer's workload. The consultant would install the trace capture tool on the customer's UNIX server, enable tracing, and would monitor the customer's system as it performs its normal duties. The consultant would then use the resulting trace to experiment with system tuning parameters, hardware upgrades, workload redistribution, *etc.*, off-line using analytical models, simulation, and perhaps also test hardware. Such traces could also be used for benchmarking and in the operating system and file system research community.

In order for this scenario to be realistic, trace capture must

- incur minimum risk and interference to the target system.
- must provide enough information for the performance tuning mechanisms to be exercised properly.
- must lead to results having adequate predictive accuracy.

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 compare, evaluate or predict the performance of the workload under different system configurations. The term *rerun* is used to describe this process. We distinguish two modes of rerunning traces: trace *replay* and trace *reexecution*. These are described below.

1.1 Trace replay

Here, we use a trace of system calls, their parameters, and fine-grain timing of the user-mode CPU times between returning from a call and issuing the next.

The trace is used to exercise a system under test using a "spinner" program. The spinner issues each call in the trace in turn, and simulates CPU time used by the application between system calls by 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.

1.2 Trace reexecution

In some applications, the spinner leads to inaccurate results because the actual behaviour of the application interacts with the operating system, for example by causing TLB misses or page faults, or by flushing operating system data from hardware caches. We can capture this by rerunning the application code.

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

The trace needed here is simpler; no timestamps are needed. System call results must be recorded, but the parameters need not.

Unfortunately, certain behaviours cannot be replayed reproducibly at reasonable cost. There are problems with asynchronous signals, and pre-emptively–

scheduled threads, which can be solved in principle by modifying the application's code (see Section 8.1). Parallel threads, and processes which interact via shared memory, are probably not reexecutable.

1.3 Time measurements

In the description above, timestamps are used to account for CPU time used by the application. There is another role for timestamps: to account for external stimuli which occur at specific wall-clock times or intervals.

To reproduce real workloads properly, it is vital to distinguish such workload-determined timing from the implementation-determined timing which is expected to vary when the configuration of the system under test is modified.

In our experiments, we assume no external stimuli with workload-determined timing. For a network server, for example, the effect of this is that the number of transactions per second is increased in proportion to the system's performance. It is reasonable, but more difficult, to keep the transaction processing rate constant and to optimise the response time.

1.4 Overview of the paper

The next section reviews some earlier contributions in the area. Section 3 describes the design of ULTra, our trace capture tool, showing how efficiency is achieved and how replay and reexecution are organised. Section 4 describes various subtleties of our implementation. The overheads of trace capture are evaluated in Section 5. Section 6 shows how accurately replay and reexecution track the application's original execution time. Section 7 presents a small case study demonstrating the predictive accuracy of the tool is evaluating the performance benefits of different amounts of RAM in WWW server application.

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. The tracing process is able to examine or modify the arguments to, and the results from, each system call. However, as noted in Section 5.1, this mechanism incurs large overheads.

Jones [6, 7] describes a general technique for interposing agents between an application and the operating system. Jones's reported work relied on an operating system facility to redirect system calls to a specified handler. Jones does not report any work on using buffering to reduce the overheads incurred by writing the trace file at each call.

Ashton and Penny [1] developed INMON, an "interaction network monitor". INMON is 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. Traces have been used extensively to study file system activity by Ousterhout et al. [8] and Baker et al. [2] in the analysis of the 4.2BSD, and Sprite distributed file systems, respectively. Bozman et al. [5] modified a CMS monitor, CMON, to gather traces of file reference patterns. Of more interest is DFSTrace, used by Mummert and Satyanarayanan [11] 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 [13] and Blaze [4] exploited a remote file system architecture to obtain traces of file system activity by monitoring the interactions between clients and server. This has the virtue of being entirely non-intrusive, though includes only remote file accesses and also requires privileged access to the network.

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.

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 [9]) for debugging purposes. Note, though, that we need to be able to reproduce the original execution time as accurately as possible.

Finally, Bitar [3] gives a useful review of the validity issues in trace-driven simulation of concurrent systems.

3 DESIGN OF ULTra

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

1. an efficient mechanism for intercepting the work-

load's system calls.

2. buffering of trace output to reduce the number of additional `write` operations incurred.

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. The advantage of this is that trace capture is confined to the library, and is therefore transparent to applications. It should be noted that although applications do not need to be recompiled, they must be relinked: however, as in modern systems the final binding between an application and a library does not occur until runtime, most applications can be traced as they are. Exceptions include rare, statically-linked applications.

For trace reexecution, we can choose how much information is included in the trace itself, and how much is accessed via the filesystem during reexecution. It is unattractive to have to include all the data the process reads, although sometimes this is unavoidable. For example, data from terminals or sockets are not available at reexecution time. Similarly, data which are overwritten later must be saved. At present, we do not log socket contents, relying instead on reexecution of the correspondent process. Nor are copies of file data included in the traces. This is adequate for our purposes.

3.1 Rerunning System Calls

On rerun the actions taken in response to a system call are determined by the captured trace, and also by the type of the system call. These fall into the following categories:

- Simple calls. In this case the responses are completely determined from the trace. Although the call need not be reexecuted to ensure the application's original behaviour is preserved, sometimes this may be necessary so as to account for the time spent servicing the call. Examples of this type of call include `getpid()` and `gettimeofday()`.

- Calls that may be rerun as before. An example of this type of call is `dup()`, which modifies the process's file descriptor table. Clearly, as this effect must be reproduced, the call must be repeated. The new return value should be identical to that in the trace. In general, the calls that fall into this category are those that modify the process's kernel state.

- Calls that must be reexecuted for their effects, but where the returned value from a replayed call may differ from that in the trace. This can occur where a system call returns a kernel-created identifier or handle for some resource that is used in later calls to identify that resource. Both trace replay and reexecution are affected, as there is no way of ensuring that the repeated call returns the same value. This is solved with the use of a table mapping capture time identifiers to those of trace rerun. An example of a call of this type is `wait()`.

3.2 Measuring Time

It is important when a trace is rerun that the system calls are reissued at the correct rate. This happens naturally in the case of trace reexecution. However, in the case of trace replay the time spent by the application executing between system calls must be simulated by the "spinner". Consequently, the trace must include the time spent executing at user level between system calls. In selecting or designing a mechanism for capturing these times the following issues must be considered:

1. the time taken to read the clock. This should be small in order to reduce the overhead of trace capture.

2. the resolution of the times reported. These should be sufficiently high to reflect the application's behaviour accurately .

3. the means by which user level execution time is identified.

4. the efficiency of the method used to communicate the times from the kernel to ULTra.

An obvious candidate for collecting these times is the resource utilization information maintained by the kernel for purposes of management or accounting (`getrusage()` or `times()`). However, the resolution of these times is that of the clock interrupt interval, typically 10–20mS, which is too coarse for our purposes.

Another alternative is to approximate the user level execution time between system calls by elapsed, 'wall-clock', time, for example, as reported by the `gettimeofday()` system call. The resolution of this time is hardware dependent, though it is often genuinely of microsecond granularity. This, like `getrusage()` above, requires two additional system calls for each call made by the application. A more important weakness is that the measured time will include time spent on other activities, for example, system activity on behalf of the process, or executing other

processes. Thus, this approach can be used only where the principal activity in the system is the application being traced. Nonetheless, when this is the case, this method can yield useful results.

Accounting for pre-emption. 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 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 a substitute for the `libc` (version 5.3.12) shared library under LINUX 2.0.25. We have also developed a statically-linked implementation for SUNOS 4.3.1.

4.1 Measuring Time

The LINUX system call mechanism was modified to include the 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 instruciton (`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. In all, the modifications were modest, amounting to about 300 lines of `C` and Pentium assembler.

4.2 Buffering

In a naïve implementation, trace records would be written out immediately. Doing so would double the number of real system calls made by an application, leading to poor performance, and consequently buffering is used to reduce the overhead. Surprisingly, buffering is ULTra's main source of complexity.

The problems affect process creation, where the actions of the new process and its parent must be coordinated to prevent corruption of the buffer or loss of trace information; and also program invocation in which the process's user level context is completely replaced, with consequential loss of the contents of the buffer. Trace capture, reexecution, and replay are all affected, but there is insufficient space to explain the details here.

5 PERFORMANCE OF ULTra

The overheads incurred by trace capture must be minimal if ULTra is to be used as we intend. In this section we present an estimate of the maximum overhead likely to be experienced (a program loops calling a system call which itself takes very little time), and also the overhead likely to be seen in more realistic applications.

All times reported in this section were obtained using a statically linked instance of version 1.7 of the GNU standard UNIX timing utility, `/usr/bin/time`. The tests were run on an unloaded IBM-compatible PC with a 166MHz Intel Pentium CPU, 32MB EDO RAM and 512KB pipeline burst-mode secondary cache, running LINUX 2.0.25. All application file input and output was to a local disk, with ULTra traffic directed to a second, local disk.

The experiments described in this section used the following applications:

- `getpid`. This is a simple program that loops calling the `getpid()` system call 1,000,000 times.
- LaTeX. LaTeX(version 2ε) is used to format a 168 page thesis.
- `apache`. The `apache` HTTP server (version 1.2b6) was configured to manage a copy of the 11,110 files (approximately 175MB) managed by our `WWW` server. In each run the server processed 25,000 HTTP requests, delivering approximately 238MB of data. The HTTP requests were derived from the access logs of our `WWW` server. In order to make the experiment repeatable for the purposes of this paper, the GET requests were issued by a simple process running on the same CPU. (We return to this example in Section 7).
- `make`. In this experiment `make` was used to recompile one version of the ULTra library. This consists of approximately 100 small files, and about 400 separate processes were involved.

The application binaries were either those distributed with LINUX, or were built from source using the default configuration and `make` options. Where necessary, the applications were compiled using version 2.7.2 of the GNU `C` compiler, `gcc` and linked to version 5.3.12 of the GNU standard library, `glibc`.

Application		Elapsed times (secs)	% of untraced time
`getpid`	untraced	2.0	100.0%
	ULTra—reexecution	4.8	242.7%
	ULTra—`gettimeofday`	13.4	675.8%
	ULTra—`rdtsc`	7.3	366.7%
	`strace`	227.2	11487.4%
LaTeX	untraced	7.6	100.0%
	ULTra—reexecution	7.5	99.6%
	ULTra—`gettimeofday`	7.6	100.6%
	ULTra—`rdtsc`	7.6	101.1%
	`strace`	9.4	123.8%
`apache`	untraced	418.7	100.0%
	ULTra—reexecution	449.6	107.4%
	ULTra—`gettimeofday`	493.5	117.9%
	ULTra—`rdtsc`	490.4	117.1%
	`strace`	985.3	235.3%
`make`	untraced	69.1	100.0%
	ULTra—reexecution	74.3	107.6%
	ULTra—`gettimeofday`	76.6	110.9%
	ULTra—`rdtsc`	77.6	112.4%
	`strace`	148.0	214.2%

Table 1: Trace capture overheads

Application		Elapsed times (secs)	% of untraced time
`getpid`	untraced	2.0	100.0%
	rerun—reexecution	4.7	237.7%
	rerun—`gettimeofday`	10.8	548.4%
	rerun—`rdtsc`	9.6	486.5%
LaTeX	untraced	7.6	100.0%
	rerun—reexecution	7.5	99.3%
	rerun—`gettimeofday`	7.5	99.3%
	rerun—`rdtsc`	7.6	99.9%
`apache`	untraced	418.7	100.0%
	rerun—reexecution	454.8	108.6%
	rerun—`gettimeofday`	480.3	114.7%
	rerun—`rdtsc`	477.2	114.0%
`make`	untraced	69.1	100.0%
	rerun—reexecution	74.3	107.6%
	rerun—`gettimeofday`	82.7	119.7%
	rerun—`rdtsc`	80.0	115.8%

Table 2: Trace replay and reexecution with unchanged configuration

In this section we consider three variants of ULTra:

1. ULTra (for replay): the traces captured include system call parameters and the user level inter-system call execution times needed by the "spinner". Execution times were approximated using `gettimeofday()`.
2. ULTra (for replay): as number 1 above, but where user level inter-system call execution time was measured using the modified kernel and `rdtsc`.
3. ULTra (for reexecution): the traces captured include system call results only. This is sufficient for reexecution.

5.1 Trace capture overheads

Table 1 shows the execution times without tracing, and with tracing for replay and for reexecution. It is unlikely that any useful application would suffer the overheads seen with the `getpid` program. The additional time is much larger for replay because of the need to gather and record timing information. It should be noted that the `rdtsc` figure is considerably better than that for `gettimeofday()`, demonstrating the efficiency of our timing and kernel to user level communication mechanisms.

The overheads for reexecution are much smaller, reflecting only the cost of copying information into the trace file and periodically issuing a `write` when it fills.

For comparison, the `strace` utility, which uses UNIX's `ptrace` mechanism, took more than 200 seconds for `getpid` (an over 100-fold slowdown), and 9.4 seconds for the LaTeX benchmark (123% of the untraced execution time). On the `apache` and `make` benchmarkes the `strace` overheads are larger, at 235% and 214%, respectively.

5.2 Buffering

We measured the effect of buffering on ULTra's performance using the `getpid` application. The unbuffered version executed in 21.93 seconds, whilst with buffering this improved to 4.8 seconds. Much of ULTra's complexity is due to buffering, and this is clearly worthwhile.

6 REPLAY AND REEXECUTION

Table 2 shows how replay and execution times compare with the original execution time for each benchmark. The replay time for the LaTeX experiment is extremely similar, indicating that paging and cache effects were negligible in the experiments, that our timing measurements are sufficiently accurate, and that our timing loops are well-calibrated. The time to replay the `getpid` experiment is disappointingly high, probably because of the overheads of reading, accessing and checking the trace. The replay times for the `apache` and `make` experiments are reasonably close, but there is room for improvement.

As expected, reexecution gives better results.

7 USING ULTra TRACES TO PREDICT PERFORMANCE

More interesting is to see how well performance on a different configuration can be predicted. To illustrate

this, we focus on the `apache` benchmark program. This 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.

7.1 Experimental design

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. For this experiment, 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).

An additional `apache` benchmark. To illustrate a richer range of behaviours, we include an additional workload for `apache` with higher RAM demand. In this variant the server was configured to manage about 4,900 documents, amounting to approximately 32MB. A list of queries was constructed such that each document was accessed twice. This was then randomly permuted and used as the workload for the experiment.

Configuration modification. Once a reexecutable ULTra trace has been captured, there are many performance analysis and tuning opportunities. As a very simple example to demonstrate the principle, we have looked at the effect of differing amounts of RAM on the effectiveness of file system caching. 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 replay trace was used for each memory size, captured from a run with the minimum 8MB configuration.

7.2 Results

Figure 1 shows the actual execution time of the original `apache` experiment for various amounts of RAM, compared with the execution time predicted by replay and reexecution of an ULTra trace captured from an original execution with 8MB RAM. In Figure 2 we show the actual and predicted execution times for the artificial workload example.

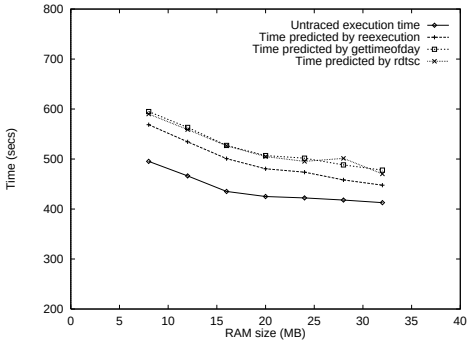

Figure 1: `apache` performance with varying RAM - predicted and actual

The execution time predicted by replaying the trace (using measured time for user mode execution, not reexecution) is within 14% for large RAM configurations, but is less accurate with small amounts of RAM where paging of `apache`'s code and/or data occurs.

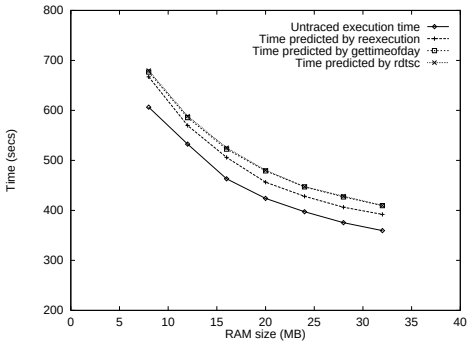

Figure 2: `apache` performance with an artificial workload and varying RAM - predicted and actual

The execution time predicted by reexecuting the trace is more accurate in all cases, and is within 10% for larger RAM configurations. This higher accuracy is because the same memory access pattern occurs during reexecution, leading to similar paging and hardware cache effects.

8 CONCLUSIONS

We have presented the design of ULTra, an efficient, portable technique for capturing traces of system call activity of a UNIX process and the processes it forks. 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 some implementation issues, which in some cases turn out to be surprisingly tricky.

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.

We evaluate two ways of rerunning a workload: replay, and reexecution. For applications where paging is insignificant, both predict performance well. Reexecution has lower trace capture overheads, and can be used to study paging, cache effects and other lower-level issues.

8.1 Further work

Asynchronous signals. Asynchronous signals can be workload-determined or implementation-determined (see Section 1.3). 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. Care is needed during trace replay to ensure that the signalled process blocks until the event it's waiting for occurs. This is necessary to ensure the replayed behaviour is consistent with the trace, but is inaccurate since the blocking is an artifact of the replay mechanism. However, in many applications the process will be sleeping (*e.g.*, when waiting for a timeout) anyway. For reexecution, it is vital for the signal to be delivered at precisely the same instruction execution point as during trace capture. The only way we know to do this (see [10]) is to modify the application's code (by recompiling or post-processing the executable). Code is added to count branches and trap on overflow. The counter is preloaded on reexecution so that the trap occurs in the basic block where the process was interrupted at trace capture time.

Pre-emptive threads. Pre-emptively scheduled threads can be handled by a similar mechanism as asynchronous signals. Details can be found in [12], where the performance overheads are reported to be around 10%.

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.

Acknowledgements This work was funded by the U.K. Engineering and Physical Sciences Research Council through a Research Studentship, and the CRAMP project (ref. GR/J 99117). Thanks also to Olav Beckmann.

REFERENCES

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

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

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

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

[5] 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.

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

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

[8] 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. 10^{th} ACM Symposium on Operating System Principles*, pp 15–24, Dec 1985.

[9] 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.

[10] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proc 3^{rd} International Conference Architectural Support for Programming Languages and Operating System (ASPLOS)*, pp 78–86, May 1989.

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

[12] M. Russinovitch and B. Cogswell. Replay for concurrent, non-deterministic shared-memory applications. In *Proc. ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pp 258–266, May 1996.

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