Tracing and Reexecuting Operating System Calls for Reproducible Performance Experiments

Ariel N. Burton and Paul H. J. Kelly

Department of Computing, Imperial College of Science, Technology and Medicine

180 Queen's Gate, London SW7 2BZ, United Kingdom

{anb,phjk}@doc.ic.ac.uk

November 16, 1998

Abstract

This paper shows how system call traces can be obtained with minimal interference to the system being characterised, 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.

Traces can be rerun in two ways: the operating system activity can be reproduced by simply replaying the system calls interspersed with appropriate delays. More interestingly, we also show how the resulting traces can be used to drive full, repeatable reexecution of the captured behaviour.

The paper concludes with an evaluation and comparison of the usefulness and accuracy of these techniques for predicting the performance impact of system configuration altenatives. We present two case studies, examining the effect of file system caching on a WWW server's performance, and the performance benefit of using a local disk instead of an NFS fileserver.

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 also in the OS 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
- provide enough information for the performance tuning mechanisms to be exercised properly
- lead to results having adequate predictive accuracy

This paper, which is an extended version of [8], presents a methodology which characterizes a workload by the trace of its system calls [1]. 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 application can interact with the operating system in ways other than through explicit system calls, for example, by causing

[1]This paper extends the conference version in giving results for additional benchmarks, adds an additional experiment addressing the tool's predictive power (NFS), and expands on many details and directions for further work.

TLB misses or page faults, or by flushing OS data from hardware caches. We can reproduce this behaviour 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 reproduced 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, namely, 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 organized. 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 two case studies demonstrating the accuracy of the tool in predicting the performance impact of configuration changes.

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 and `/proc` file system are examples of mechanisms provided to allow 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 issued by the traced process. However, as is noted later in Section 5.2, this approach incurs large overheads.

Jones [11, 12] describes a general technique for interposing agents between an application and the OS. One example considered is tracing system calls. Jones' reported work relied on an OS 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* [14] and Baker *et. al* [3] in the analysis of the 4.2BSD, and Sprite distributed file systems, respectively. Bozman *et. al* [6] modified a CMS monitor, CMON, to gather traces of file reference patterns. Of more interest is DFSTrace, used by Mummert and Satyanarayanan [17] 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 OS kernel, Tourigny [20] and Blaze [5] 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.

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 [19]. 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 [15]) for debugging purposes. Note, though, that we need to be able to reproduce the original execution time as accurately as possible.

Finally, Bitar [4] gives a useful review of the validity issues in trace-driven simulation of concurrent systems.

3 Design of ULTra

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

1. an efficient mechanism for intercepting 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, since there is no way of ensuring that the repeated call returns the same value. This problem 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 (*e.g.*, as reported by the `getrusage()` or `times()` system calls). However, on LINUX and many other operating systems, 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. This represents a considerable overhead. 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, where this is the case, and where pre-emption is not a concern, this method can provide useful results (see, for example, [8] for results based on `gettimeofday()`).

3.3 Accounting for Pre-emption

We account for user-level execution time of a process in the presence of other processes by modifying the kernel to update a timer in the process's process table entry on each context switch to, or from, user mode. To keep this 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 clock-cycle 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 an 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. 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 3.3. 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.

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. Program invocation, in which the process's user-level context is completely repalced, is also affected, since the contents of the buffer are overwritten and lost. 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 lightweight 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-mod 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.

- mSQL. This experiment involved running part of the AS^{3}AP[21] SQL benchmark on mSQL[9], a lightweight database engine. As mSQL implements only a subset of SQL, the AS^{3}AP benchmark suite was modified accordingly. Version 2.0.3 of mSQL was used.

  For the experiments involving this benchmark, each of the four major relations specified by AS^{3}AP included 10,000 tuples, averaging approximately 100 bytes each. Including management overheads, this amounted to approximately 8MB. In addition, during the course of the experiment, mSQL manipulated at least 15 index files, each averaging at least 0.5MB.

  In the experiments, the SQL requests were issued to the server over a UNIX domain connection by the interactive monitor distributed with mSQL. The replay and reexecution cases were handled slightly differently:

reexecution: in this case, only the server was traced. On reexecution, the monitor was reexecuted to reproduce the requests for the server.

replay: it was quickly noticed that the behaviour of the monitor depended on the responses it received from the server. On replay, although the communication link and sequence of messages could be reproduced easily, the contents of the messages could not. As a result, the monitor interpreted the replies it received from the replaying server as invalid, and attempted to recover from the 'error' condition by, *e.g.*, resending the request. This had the effect of changing the pattern of communication between the monitor and the server, and as a consequence, the new interactions did not match those in the server's trace.

This problem was solved by tracing both the server and the monitor, and driving both sides of the communication from the traces.

It should be noted that in a more comprehensive implementation this problem would not arise, as the network input to the server would have been recorded by a network snooper, and replayed from an external source. The problem described above is simply a consequence of using the monitor to replay the network inputs to the server.

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`.

In this section we consider two variants of ULTra:

1. ULTra (for reexecution): the traces captured include system call results only. This is sufficient for reexecution.

2. ULTra (for replay): the traces captured include system call parameters and the user level inter-system call execution times needed by the "spinner". User level inter-system call execution time was measured using the modified kernel and `rdtsc`.

5.1 Overheads of the Kernel Modifications

Section 4 described the modifications to the kernel that are needed to support ULTra. Some of these modifications affect all processes in the system, whether or not they are being traced. These

Application	Method	Elapsed time (secs)	% of untraced time
`getpid`	untraced	1.4	
	ULTra—reexecution	4.4	314%
	ULTra—replay	7.5	536%
	`strace`	198.5	14179%
LaTeX	untraced	7.7	
	ULTra—reexecution	8.0	104%
	ULTra—replay	7.8	101%
	`strace`	9.2	119%
`make`	untraced	92.7	
	ULTra—reexecution	100.6	109%
	ULTra—replay	101.6	110%
	`strace`	169.4	183%
mSQL	untraced	92.4	
	ULTra—reexecution	112.2	121%
	ULTra—replay	125.0	135%
	`strace`	220.0	238%
`apache`	untraced	416.2	
	ULTra—reexecution	446.8	107%
	ULTra—replay	490.4	118%
	`strace`	898.1	216%

Table 1: Trace Capture Overheads

modifications include those to the kernel entry and exit points needed to measure user-level inter-system call execution time accurately. Although this information is written to the user process's address space for only those processes that are being traced, internally this instrumentation is always active, and therefore all processes, traced or untraced, experience the overhead introduced by these modifications.

The performance overhead the kernel modifications for untraced processes has been measured for all the benchmarks presented here. For the worst case, `getpid`, execution times are increased by 7% with the kernel needed for reexecution, and 50% for the kernel needed for replay. The overheads on the realistic benchmarks are small, all under 3% [7].

5.2 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.

The increase in execution times reflect, firstly, the time taken by the kernel to copy any information to the user process's address space, and secondly, the time taken by the ULTra runtime to copy the trace information to the trace buffer, and periodically, when the buffer fills, call `write()` to dump the buffer contents to the trace file. It can be seen that in general the overheads for trace capture for trace replay are larger than those for trace reexecution. This is for two related reasons: firstly, the trace records are larger because they must include the inter-system call execution times necessary for replay. Thus more I/O is required. Secondly, as the trace records must include user-level execution times, information must be copied to user space after each system call. This component of the overhead is dominated by the need to check the validity of the destination before the data may be copied. Copying the data, on the other hand, is relatively lightweight. In contrast, the overheads for reexecution are much smaller, reflecting only the cost of managing the trace buffer and the associated I/O. This effect is most obviously seen in the `getpid` case, in which the application simply loops issuing system calls, and in which ULTra accounts for a significant proportion of the application's overall execution time.

For comparison, the `strace` utility, which uses UNIX's `ptrace()` mechanism, took just under 200 seconds for `getpid` (an over 140-fold slowdown), and 9.2 seconds for the LaTeX benchmark (119% of the untraced execution time). On the `make`, mSQL, and `apache` benchmarks, the `strace` overheads are larger, at 183%, 238%, and 216% of the untraced time, respectively.

mSQL shows very different trace capture times for reexecution and replay. The reason for this is that, as noted above, for replay it was necessary to trace both the server and the client, whereas for reexecution only the server was traced.

5.3 Buffering

We measured the effect of buffering on ULTra's performance using the `getpid` application. The unbuffered versions executed in 13.9 seconds (reexecution) and 21.7 seconds (replay), whilst with buffering this improved to 4.2 seconds, and 11.4 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 reexecution 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`

Application	Method	Elapsed time (secs)	% of untraced time
`getpid`	untraced	1.4	
	ULTra—reexecution	4.2	300%
	ULTra—replay	9.2	657%
LaTeX	untraced	7.7	
	ULTra—reexecution	7.7	100%
	ULTra—replay	7.8	101%
`make`	untraced	92.7	
	ULTra—reexecution	103.7	112%
	ULTra—replay	104.7	113%
mSQL	untraced	92.4	
	ULTra—reexecution	116.6	126%
	ULTra—replay	103.9	112%
`apache`	untraced	416.2	
	ULTra—reexecution	454.8	109%
	ULTra—replay	477.2	115%

Table 2: Trace replay and re-execution with unchanged configuration

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. The replay time for mSQL, in which there is considerably more ULTra activity since both the client and the server are rerun from the traces, is very much lower than the time for reexecution, where only the server is traced. This is a little unexpected. The reason for this discrepancy is that as well as making system calls, mSQL causes system activity as a result of its memory accesses. This is because mSQL uses `mmap()` to map some of the files it uses into its address space. Once mapped, the files can be accessed as ordinary memory, and consequently, reads from, or writes to, this memory can cause real I/O, and therefore system activity. This activity is hidden from ULTra replay, and therefore when the the trace is replayed, this component of the workload is omitted.

As expected, overall, 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. The experiments described in this section are designed to determine ULTra's effectiveness in this role.

We evaluated ULTra for predicting the performance of two example scenarios:

1. Using an NFS-mounted file system instead of local disks.

2. Changing the amount of RAM available for caching file accesses.

These experiments are described in the following sections.

7.1 Benchmarks for Performance Prediction Experiments

ULTra is designed for workload characterization in situations where an application is interacting with its environment in complicated ways which make it difficult to redo experiments with precisely reproducible results. However, for the purposes of these experiments, in order to be able to determine the accuracy of the predictions made by ULTra, the trace rerun execution time must be compared with the actual time taken to execute the workload on the alternative configuration.

We chose the benchmarks in order to overcome this problem. For example, the `apache` WEB server 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.)

7.2 Performance Prediction: Using an NFS-mounted File System

The performance of an application can depend very heavily on the type of file system on which its files reside. In these experiments, ULTra was used to predict the effect storing an application's files on a remote machine has on its performance. The aim of these experiments was to determine how well ULTra is able to predict this effect.

7.2.1 Experimental Design

The traces used in the experiments were those captured whilst the system was configured so that the applications' file were stored locally. The system was then reconfigured so that the files required by the applications resided on a remote machine, and were accessed over a local area network using NFS. The server used for this purpose was another PC (233MHz Intel PENTIUM II with 128MB on a 10Mb/s Ethernet, running LINUX 2.0.30). For the purposes of the experiments, the server was unloaded, and other network traffic was eliminated by ensuring that only the client and server were connected to the network. In this experiment, the following applications were considered, and configured as described:

`make` in this experiment, the machine was configured so that the source and temporary files needed for this workload resided on the server. Other files, such as the standard headers included

Application	Method	Elapsed time (secs)	% of actual time
`make`	actual	137.7	
	predicted—reexecution	144.4	105%
	predicted—replay	148.1	108%
mSQL	actual	184.7	
	predicted—reexecution	196.8	107%
	predicted—replay	120.6	65%

Table 3: Predicted execution times when the benchmarks' files are located on a remote volume, and must be accessed over a network

by many program source files were held and accessed locally.

mSQL in this case, the database and index files manipulated by this application were stored on the server.

In all cases, the traces used for the prediction phase of the experiment were those captured for the experiments to determine the overhead of trace capture (see Section 5.2), and in which all files were stored locally. In addition, in order to determine the accuracy of the prediction made by ULTra, the application was also executed on the alternative configuration.

7.2.2 Results

Table 3 shows the actual execution times for these applications on this configuration, as well as those predicted by trace replay and trace reexecution. In general, the ULTra predictions are acceptable. However, of note is the replay prediction for mSQL. Here, ULTra has significantly underestimated the execution time achieved by this application. The reason for this discrepancy is that, as noted earlier, mSQL uses `mmap()` to map some of its files into its address space. Once mapped, it accesses the files as ordinary memory, although in doing so it will cause network I/O and system activity. However, trace replay is unable to reproduce mSQL's memory accesses, and therefore the network I/O that would ensue. We return to this issue in Section 8.1.

7.3 Performance Prediction: Changing the Amount of RAM Available

In this experiment 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 our 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.

7.3.1 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.

7.3.2 Experimental Design

Once a rerunable 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 both to replay an ULTra trace, and also to reexecute a trace. The same traces were used for each alternative memory size; these were captured from runs with the minimum 8MB configuration.

7.3.3 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.

The execution time predicted by replaying the trace (using measured time for user-mode execution, rather than 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.

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.

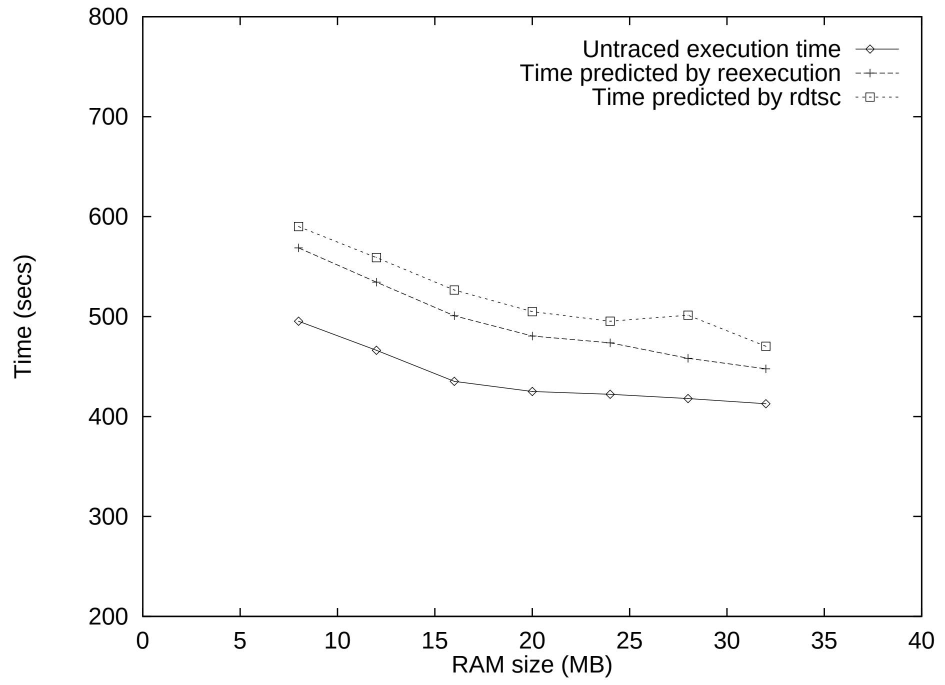

Figure 1: `apache` performance with varying RAM—predicted and actual

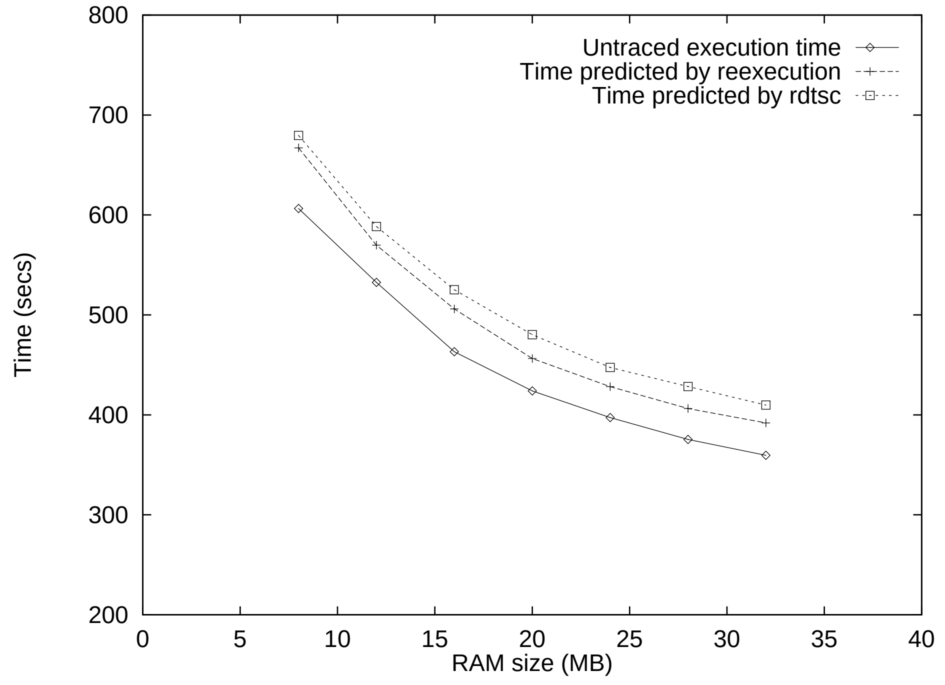

Figure 2: `apache` performance with an artificial workload and varying RAM—predicted and actual

8 Conclusions and Discussion

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 certain 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 two case in which this is illustrated. We demonstrate that ULTra can be used to capture a trace of the workload without substantial interference, which can then 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.

In the performance prediction experiments presented here, we were able to compare the behaviour with reproducing the workload by other, more straightforward, means. The results in this paper provide evidence to support the use of ULTra in situations where such validation is not possible, for example, where the alternative configuration is being simulated, but the simulation slowdown could lead to a change in user behaviour.

8.1 Further work

Capture paging activity. Trace replay is potentially inaccurate compared with reexecution because it does not capture paging behaviour (resulting, for example, from mSQL's use of `mmap` as seen in Section 7.2.2). Although it may be possible to intercept and trace paging events, the behaviour on a different configuration may be very different. We are working on introducing additional instrumentation to use page protection to track the process's memory access behaviour so that we will be able to predict the paging behaviour with various amounts of RAM and with different virtual memory management policies. Preliminary results are very promising.

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, could easily be traced. Care is needed during trace replay to ensure that the signalled process blocks until the event for which it's waiting 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 anyway (*e.g.*, when waiting for a timeout).

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 [16]) is to modify the application's code (by recompiling or post-processing the executable). Code is added to count backward 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 [18], where the performance overheads are reported to be around 10%.

Interaction *via* shared-memory regions. Perhaps the most intractable problem is to trace processes which interact *via* a shared memory region. In principle, Unix processes on a single Cpu could be dealt with as indicated above for pre-emptive threads, but with a shared-memory multiprocessor there appears to be no solution with reasonable overheads (though see, for example, [13, 2]).

Tracing dynamic linking itself. Since we intercept system calls *via* a substitute dynamically-linked library, we cannot trace the dynamic linking mechanism itself (nor statically-linked programs or programs which bypass the library and trap to the OS directly). OS-level tracing (*e.g.*, using `trace`) does manage this, and one option is to use `trace` during reexecution of the ULTra trace.

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).

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] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *ACM/ONR Workshop Parallel and Distributed Debugging*, Santa Cruz, CA (USA), May 1991.

[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] 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.

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

[6] 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.

[7] A. N. Burton. *The Use of Replayable Traces in the Design and Evaluation of Operating Systems.* PhD thesis, Imperial College of Science, Technology and Medicine, Department of Computing, London, United Kingdom, 1998.

[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] Hughes Technologies Pty Ltd, P.O. Box 432, Main Beach, Queensland 4217, Australia. *Mini SQL User Guide*, 2.0v1 edition, July 1997.

[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] P. Keleher and W. Zwaenepoel. Detecting data races in software distributed shared memory. Technical Report TR91-171, Deptarment of Computer Science, Rice University, Houston, Texas, Nov. 1991.

[14] 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.

[15] 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.

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

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

[18] 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*, pages 258–266, May 1996.

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

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

[21] C. Turbyfill, C. Orji, and D. Bitton. AS^{3}AP: An ANSI SQL standard scaleable and portable benchmark for relational database systems. In J. Gray, editor, *The Benchmark Handbook: for database and transaction processing*, chapter 4, pages 167–206. Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, Suite 260, SanMateo, CA 94403, USA, 1991.

Ariel Burton

Ariel Burton currently holds a teaching position at the Department of Computing in Imperial College of Science, Technology and Medicine in London, UK. His research interests include operating systems, performance evaluation, and parallel and distributed systems. He received a BSc (Eng) in Computing Science from Imperial College (1990), an MSc in Advanced Methods in Computer Science from Queen Mary and Westfield College (1992) and a PhD from Imperial College (1998). He can be reached at Imperial College of Science, Technology and Medicine, Department of Computing, London, SW7 2BZ; `anb@doc.ic.ac.uk`.

Paul Kelly

Paul H J Kelly is a Senior Lecturer in the Department of Computing at Imperial College of Science, Technology and Medicine in London, UK, where he leads a research group in high-performance computer systems, architectures, compilers and programming languages. He received a BSc in Computer Science from University College, London in 1983, and a PhD in parallel functional programming from Westfield College, also part of the University of London, in 1987. His email address is `p.kelly@doc.ic.ac.uk`.