Performance prediction of paging workloads using lightweight tracing

Ariel N Burton[1]
Paul H J Kelly
Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK
p.kelly@imperial.ac.uk

Abstract

A trace of a workload's system calls can be obtained with minimal interference, and can be used to drive repeatable experiments to evaluate system configuration alternatives. Replaying system call traces alone sometimes leads to inaccurate predictions because paging, and access to memory-mapped files, are not modelled.

This paper extends tracing to handle such workloads. At trace capture time, the application's page-level virtual memory access is monitored. The size of the page access trace, and capture overheads, are reduced by excluding recently-accessed pages. This leads to a slight loss of accuracy. Using a suite of memory-intensive applications, we evaluate the capture overhead and measure the predictive accuracy of the approach.

1 Introduction

We aim to develop a tool for a system performance consultant to characterize workloads that are complex and subject to external influences and stimuli which cannot be controlled 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 to rerunning traces: trace *replay* and trace *reexecution*:

- **System call trace replay**

  The trace contains descriptions of each of the system calls issued by the application, including parameters, and fine-grained timings of the application's (user-mode) execution times between system calls. The system under test is exercised by re-issuing the system calls. The application's activity is modelled by looping for the appropriate amount of time between issuing each call.

- **System call trace trace reexecution**

  In some circumstances looping to account for user-mode execution time leads to inaccurate results because the application interacts with the operating system implicitly, for example, by causing TLB misses, page faults, or by flushing operating system code and data from the hardware caches. We can reproduce this behaviour by reexecuting the original application code.

  In order for the original application code to behave as it did originally, the results returned from the application's system calls are recorded in the trace. The reexecuting application code should behave in a precisely reproducible way since it is fed precisely the same inputs.

Trace reexecution should give more accurate performance predictions than trace replay because a workload's behaviour is reproduced more completely. However, not all applications are easily reexecuted. Threads and asynchronous signals present difficulties. Shared-memory regions and multi-processor applications are extremely hard to trace efficiently.

Furthermore, confidentiality or size constraints may forbid the workload's input data from being copied and used during the performance analysis.

In such cases, the only trace rerun option available is the potentially less accurate trace replay. Our earlier work [3, 4] focused on workloads which interact with the operating system predominately through the system call interface. These workloads were adequately characterized by trace replay. In contrast, in this paper we examine workloads which exer-

[1] Ariel Burton is now with Etnus, Inc.

cise the operating system as a consequence of their memory referencing behaviour. We show how our user-level trace capture tools can be extended to include paging information.

It would be easy to log the actual page faults that occur. However, this fails to characterise the application's behaviour - what if more, or less RAM were available to this process? Instead, we need to log the application's memory access pattern – in sufficient detail for paging behaviour on different configurations to be faithfully predicted.

With this information recorded in the traces, trace replay can be extended to reproduce a workload's memory referencing behaviour at the page level, thereby improving the accuracy of the predictions that can be made with this method.

1.1 Contributions of this paper

1. We evaluate the idea of recording a computer system's workload for later replay by tracing system calls, and measure the impact of trace capture on the application's performance
2. Using a suite of example applications, we explore how faithfully performance of alternative system configurations can be predicted using recorded workload traces
3. We present a technique for enhancing system call tracing to capture an application's paging behaviour, and evaluate the capture overheads and predictive value of the approach.

Our earlier paper [4] (an extended version of [3]) explores system call tracing for workload characterisation. This paper extends this to include paging behaviour.

2 Related work

Trace capture is a well established technique for performance evaluation. The critical aspect of our work lies in identifying and capturing the very minimum information, *i.e.*, system calls, to be able to reconstruct the original workload by reexecution. This means a real, interactive workload can be recorded — then repeatably replayed.

Jones [7] describes a general technique for interposing agents between an application and the operating system. One example presented is tracing system calls. DynInst [5], and the ptrace system call offer a similar capability.

CASPER, a general-purpose trace capture package for kernel developers, was used to improve the performance of individual system calls [2] and also to obtain samples of processes' memory references. Ashton and Perry [1] developed INMON, an "interaction network monitor" for distributed systems. INMON is designed to follow the flow of control as it passes from process to process, kernel to kernel, and machine to machine in response to individual user actions.

To capture a workload's memory access pattern, many researchers have traced memory access instructions, either using a simulator or by augmenting binary code. Using control-flow analysis the quantity of information generated at trace capture time can be reduced substantially. The full traces are recreated by combining the captured trace with the original application [8].

3 Background: system call tracing

For our approach to be viable and attractive, the tool must incur minimum risk to the system under examination, provide enough information for performance tuning mechanisms to be exercised properly, and lead to results having adequate predictive accuracy. In this section we focus on the issues of trace capture, trace reexecution, and trace replay. For further details see [4].

3.1 Trace capture

ULTra (User Level TRAcing) intercepts system calls and writes trace information to a trace file. Having considered various alternatives, we chose to substitute the dynamically-linked standard shared library implementing the UNIX system call interface. In the ULTra version, the system call stubs are extended with modifications for trace capture.

3.2 Trace reexecution

For re-execution to work we must recreate the workload's environment from its traces. The workload's system calls will be reissued naturally as the application code is re-executed, but the values returned 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()`). Calls of this type are handled by keeping a translation table mapping trace capture values to trace reexecution values.

3.3 Trace replay

In this form of trace rerun we reissue the system calls made by the original workload. We simulate the user-level execution times between system calls by looping for the appropriate period. We use the same techniques as trace reexecution to handle translation of trace capture identifiers to trace replay identifiers.

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-mode inter-system call execution times. This essential information is not provided in standard UNIX implementations.

We account for a process's user execution time in the presence of other processes by modifying the kernel to update a timer (using the hardware timestamp counter) in its process table entry on each context switch to, or from, user mode.

4 Capturing paging

The challenge in extending ULTra replay to reproduce a workload's paging activity is to do it efficiently, at user level, and with unmodified applications.

4.1 A simple algorithm

Figure 1 outlines a simple algorithm for capturing an application's page-level memory access at user level (we improve on this shortly). The ULTra runtime library is modified so that before the application is allowed to start, a signal handler is installed to catch invalid memory references (step 1). In addition, the permissions to the regions in the process's address space are modified to disable access (step 2). When the application starts (step 3), its first memory reference will cause a fault to be raised (step 4). This will be caught by the handler installed in step 1. The handler identifies the address and access mode of the reference which caused the fault, and records this information in the trace. Before returning to the application (step 7), the handler enables access to the page on which the fault was raised (step 5). This allows the application to continue until it attempts to access a different page. When this occurs, a fault is raised as before. This is handled in the manner described above, with the exception that before allowing the application to continue, access to the previously enabled page is disabled (step 6).

This simple algorithm illustrates how a full trace of an application's page referencing behaviour can be captured entirely at user level. However:

1. The overheads of trace capture are very large, since a signal must be raised every time the application refers to a new page.
2. The traces will be very large, since a record is written for each page used by the application. Dealing with the inflated traces will further increase the overheads of trace capture.

These problems can be overcome by applying a trace reduction technique originally developed to reduce the number of records which must be examined during the analysis of a full trace.

1. Install a signal handler to catch invalid memory references.
2. Modify the permissions of all the regions in the process's address space to deny access. This will cause a fault to be raised when a protected page is accessed.
3. Allow the application to start.
4. When a fault occurs, record the address and access type which caused the fault in the ULTra trace.
5. Enable access to the page on which the fault was raised.
6. Disable access to the currently enabled page.
7. Return to the application.

Figure 1. Simple algorithm for tracing page-level memory accesses at user level

4.2 Trace reduction using stack deletion

Smith [12] proposed a method called Stack Deletion to reduce the cost of analyzing memory trace data. Stack Deletion exploits the principle of locality to reduce the number of trace records which must be examined during trace analysis.

The reduced memory traced is obtained by using the full trace to drive a simulated LRU stack memory. Each time a page is referenced, the page is moved to the top of the stack, and all elements above its former location descend one position. For trace deletion parameter D, references which hit in the top D levels of the stack are deleted from the trace. The reduced trace consists of the remaining references which miss the top D entries.

The reduced trace describes every entry into the locality defined by the D most recently used pages. However, since references which hit the top of the stack are deleted, all information about the usage history of these pages is lost. To see how this affects the accuracy of the subsequent analysis, consider the top $D-1$ elements of the memory stack that results from the reduced trace. The element in position $D-1$ could really belong in position 1 of the true LRU stack, and *vice versa*. More generally, at any time during the analysis, the order of the top $D-1$ elements of the memory stack obtained using the reduced trace will be some permutation of the true order. This affects the order of the rest of the stack, since the entry at position i is either at position $i+1$ or position 1 after the following trace record has been processed. One consequence of this is that the elements expelled from a simulated memory of a given size will be different for the full and reduced traces. Smith [12] compared the fault rates predicted by the reduced and full traces for a variety

of page replacement policies, including LRU and CLOCK, and found the error introduced by incompleteness of the information in the reduced trace to be less than 5%, and that Stack Deletion reduced the length of the trace by up to two orders of magnitude.

4.3 Applying stack deletion to improve the performance of trace capture

In this section we use Stack Deletion to improve the performance of trace capture.

We extend the ULTra library to follow the simple algorithm described earlier, but with the exception that the application is allowed access to a set of pages, rather than just one. The function of this pool of pages is the same as that of the LRU stack memory: to filter out intermediate references to the locality represented by the set of pages.

In order to mimic stack deletion faithfully, the page that is discarded from the pool should be the one that has been least recently used. This is easy when stack deletion is used to reduce a full trace, but more difficult in our case because the information needed to determine the identity of the least recently used page is not available. This is because the intermediate references which define the usage history of the pages in the pool do not cause faults, and therefore they are not visible to ULTra. Although the operating system should have some information about the process's memory usage, there are usually no facilities for obtaining this information. Thus, we cannot use LRU as the replacement policy for the pages in the filter.

Instead, we use FIFO, and expel the page which entered the filter first. Poon [10] demonstrated that the numbers of faults predicted by reduced traces derived from a FIFO filter were within 2% of the corresponding numbers predicted by the full traces for fully-associative memories implementing LRU and CLOCK replacement policies. This is not unexpected, since FIFO does not do as well as LRU in keeping recently used pages in the filter. Consequently, the number of misses from a FIFO filter would be expected to be larger than the number from a filter using LRU. The effect of this is that FIFO does not reduce the size of the trace as well as LRU, but the trace does contain more information.

4.4 Replaying paging behaviour

As before, the "spinner" reads the trace record by record. System calls are treated in the same way as ordinary trace replay. When a memory reference is encountered, the "spinner" either reads from, or writes to, the location specified, thus causing the page to be touched in the same way as at the time of trace capture.

5 Implementation

ULTra is currently implemented as two major 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.

5.1 Kernel modifications

The LINUX system call mechanism was modified to include the time measurement extensions described in Section 3.3. We use the PENTIUM processor's 64 bit Time Stamp counter to determine the number of clock cycles a process spends executing at user level. The user-level execution times are communicated to the user-level component of ULTra using a memory region at the base of the stack. In all, the modifications were modest, amounting to about 200 lines of `C` and PENTIUM assembler.

5.2 Implementing extended ULTra

Implementing extended ULTra is relatively straightforward, though there were a number of issues, mainly affecting trace capture, which must be handled extremely carefully. Although there is insufficient space to describe these in detail, the salient points are summarized below.

Excluding ULTra's memory activity Memory activity caused by ULTra must be excluded from the trace. The memory used by ULTra falls into two categories: memory private to ULTra, *e.g.*, the trace buffer and its management routines, and memory used by both ULTra and the application, *e.g.*, standard library routines.

For reasons of efficiency, accesses by ULTra to its private code and data should not raise faults. This is achieved by excluding these areas from the memory to which access is disabled during initialization. Determining the addresses from `C` level is easy, though care is needed since the compiler or linker may rearrange the memory map. In addition, the indirection tables required to locate position independent dynamically linked code and data must also be handled carefully.

Memory used by both ULTra and the application cannot be handled in this way, since none of the application's references will appear in the trace. To ensure that ULTra's references are excluded, a counter is incremented on every entry into, and decremented on every exit from, ULTra. When a fault is raised, the counter is inspected by the handler; if the fault was caused by ULTra, access to the page is enabled, but the page is not added to the filter, and instead to a separate data structure. On the last exit from ULTra, access to any pages in this page is disabled.

System calls which pass arguments by reference A number of system calls accept arguments or return results by reference. These calls will fail where they might otherwise succeed if access to the parameters has been disabled by ULTra. The system call is allowed to proceed, but if it fails because the parameters could not be accessed, access is enabled and the system call retried.

Modifications to the address space The UNIX API provides a number of system calls which allow a process to manipulate its address space. These system calls include operations to add or remove regions, as well as change their access protections. In general, when additions are made to the process's address space, access to the new areas is disabled. When regions are removed, all pages from that region which are present in the filter are removed.

6 Experimental evaluation

The experiments reported here were performed 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 (or variants thereof). All application file input and output was to a local disk, with ULTra traffic directed to a second, local disk. Elapsed execution times were measured using a statically linked instance of version 1.7 of the GNU standard UNIX timing utility `/usr/bin/time`.

6.1 The benchmark suite

The experiments used the following applications:

- `qsort`. An in-core sort of 2,000,000 pseudo-random integers using the `qsort()` function supplied with the `C` standard library.
- C4.5. Given a training set and a set of pre-defined classes, C4.5 [11] attempts to generate a function which maps the data items in a database into the pre-defined classes, by constructing an optimal decision tree. The data set used in these experiments was the 'Connect-4 Opening Database' [9].
- mSQL. This experiment involved running part of the AS^{3}AP[13] SQL benchmark on version 2.0.3 of mSQL[6], a lightweight database engine. The data managed by the SQL server were generated using `as3ap_gen`, a utility written for this purpose. As mSQL implements only a subset of SQL, the AS^{3}AP benchmark suite was modified accordingly.

  For the experiments, each of the four major relations specified by AS^{3}AP included 10,000 tuples, averaging approximately 100 bytes each. Together with the management overheads introduced by mSQL, and also the overflow buffers required to store variable length fields, 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 size.

  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 rerun, the server was reexecuted from the traces. The requests were reproduced by reexecuting the interactive monitor.

  replay: The behaviour of the monitor depends on the responses it receives from the server. On replay, although the communication link and sequence of messages could be reproduced easily, the contents of the messages could not.

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

  This problem would not arise in a more comprehensive implementation, since network input would be recorded by a network snooper, and replayed from an external source. The problem described here is simply a consequence of using the monitor to replay the network inputs to the server.

These benchmarks were selected because they use memory in a variety of ways. `qsort` makes very few system calls, and because it is very memory intensive, its performance is likely to depend very heavily on the availability of RAM. C4.5 uses the file system for its first and third phases, but includes a very memory intensive second phase. mSQL was chosen because it is very complex, and interacts with the underlying system in a number of different and subtle ways.

The applications were built from source using the default `make` and compile options, using version 2.7.2 of the GNU `C` compiler, `gcc`, and linked to version 5.2.12 of the GNU standard library, `glibc`.

6.2 Overhead of trace capture

Extended ULTra is intended for use in circumstances where a workload's paging activity forms a significant part of the load on the system. For this reason, in these experiments, each of the benchmarks was traced on system configurations which experience had shown ordinary replay to be inadequate. These are are summarized in Table 1. Also shown are the sizes of the filters used in each case; in each case the filter represents about an eighth of the minimum

Application	System configuration	Filter size	
		No. pages	MB
mSQL	Database & index files local	512	2
`qsort`	RAM size set to 8MB	256	1
C4.5	RAM size set to 15MB	512	2

Page size = 4096 bytes

Table 1. System configurations for each of the benchmarks

Application	Method	Time (secs)	% of untraced time
mSQL	untraced	92.4	
	for reexecution	112.2	121%
	for ordinary replay	125.0	135%
	for extended replay	158.8	172%
`qsort`	untraced	383.3	
	for reexecution	396.3	103%
	for ordinary replay	397.3	104%
	for extended replay	414.2	108%
C4.5	untraced	250.9	
	for reexecution	485.2	193%
	for ordinary replay	771.5	307%
	for extended replay	1300.4	518%

Table 2. Overheads of trace capture

Application	Filter size		No. faults	No. records
	No. pages	MB		
mSQL	512	2	58,373	439,390
`qsort`	256	1	39,852	39,863
C4.5	512	2	3,268,012	3,273,339

Table 3. Number of faults and total number of trace records for the benchmark applications

RAM size on which the traces were to be replayed (see later).

Table 2 shows the untraced and traced execution times for each of the benchmarks and flavours of ULTra.

Capture overheads for system call tracing In general, the overheads of trace capture for ordinary replay are larger than those for trace reexecution. Firstly, the trace records are larger since they must include the inter-system call execution times necessary for replay. Secondly, these times must be copied from the `ultra_area` after each call. This component of the overhead is dominated by the need to check the validity of the destination before the data may be copied. Additionally, in mSQL, replay involved more ULTra activity since both client and server were traced. The overheads for reexecution and ordinary replay for C4.5 are interesting since earlier experiments showed overheads of approximately 1%. One reason for the increase seen here is that on small RAM configurations, the performance of C4.5 is very sensitive to the availability of memory. For example, untraced execution time at 14.5MB and 15MB are about 2400 and 250 seconds, respectively. (See also Figure 2(b).) The ULTra versions of the standard library are larger than those of the uninstrumented version, and therefore when it is traced, there is less memory available for the application.

Overheads of capturing paging The overheads for extended replay are higher than those for both reexecution and ordinary replay. This is a consequence of the considerably more ULTra activity associated with dealing with the faults, and I/O required for the much larger traces (see Table 3). The overheads for `qsort` are reasonable, though mSQL and C4.5 show large increases. This is a little surprising in the case of mSQL, where paging information accounts for a very smaller proportion of the contents of the trace than with the other benchmarks.

Examination of the mSQL traces showed that the number of pages in the filter rarely exceeded 120 (420KB), even though the application used considerably more memory. The reason for this lies in how mSQL uses its files. mSQL maps its files into its address space using `mmap()`. Once mapped, the contents of the file can be accessed using ordinary memory operations. However, the file cannot be extended, and therefore when this is necessary, mSQL unmaps the file, extends it by writing to it, and then remaps the file. When the file is unmapped, any pages from that region must be removed from the filter. In our implementation this required a sequential traversal of the array representing the filter. This behaviour also accounts for the small number of pages in the filter.

C4.5 takes a large number of faults (see Table 3). Examination of the traces showed that this is because C4.5 makes many sequential scans through its data. Thus, as C4.5 has little locality, the filter is repeatedly flushed, and overheads rise.

The results show that the overhead per fault is about 300μs. A proportion of this can be attributed to the reduced availability of memory, and to the I/O required to deal with the substantially larger traces. The remainder is the time required to take and process the fault, which further investigation showed to be about 80μs. Microbenchmarking showed that the irreducible overhead of taking a fault and calling `mprotect()` twice (once to enable access to the faulted page, and a second to disable access to the page removed from the filter) is about 45μs. The remainder is time spent

preparing the trace record and updating data structures such as a filter; this can be reduced with a more careful and optimized implementation.

7 Using ULTra to predict performance

In this section we study the value of using traces to predict performance of different configurations.

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, if we are to be able to determine the accuracy of the predictions made by ULTra, the trace rerun execution times must be compared with the actual time taken to execute the workload on the alternative configuration.

The traces used in these predictive experiments were those captured during the experiments described earlier. Then, for each alternative configuration, the trace was rerun and the application was also executed. We evaluated ULTra for two example scenarios:

1. using an NFS-mounted file system in place of a local disk;
2. changing the amount of RAM available to the workload.

We describe these experiments in the following sections.

7.1 Predicting effect of changing the 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 of storing an application's files on a remote machine has on its performance. Our aim in these experiments was to see how well each of the different forms of ULTra is able to predict this effect.

For this experiment we used the mSQL benchmark. The traces used here were those captured with the files stored locally. The system was then reconfigured so that mSQL's files resided on a remote machine, and were accessed using NFS. The server used for this purpose was an IBM-compatible PC with a 233 MHz Intel PENTIUM II CPU, 256KB level 2 cache, 128MB SDRAM, running LINUX version 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.

Table 4 shows the actual execution time achieved by mSQL on the modified configuration, and also those predicted by ULTra. Ordinary replay has significantly underestimated the execution time. The reason for this is that, as noted earlier, mSQL uses `mmap()` to map some of its files into its address space, which are then accessed as ordinary memory. In this case, however, the system activity that is caused as the files are used is related to network I/O. Ordinary replay cannot reproduce the memory accesses, and therefore the time predicted will exclude that accounted for by this I/O. Since a large proportion of mSQL's file accesses use this mechanism, the discrepancy is large.

Method	Time (secs)	% of actual time
actual	184.7	
time predicted by reexecution	196.8	107%
time predicted by ordinary replay	120.6	65%
time predicted by extended replay	189.3	102%

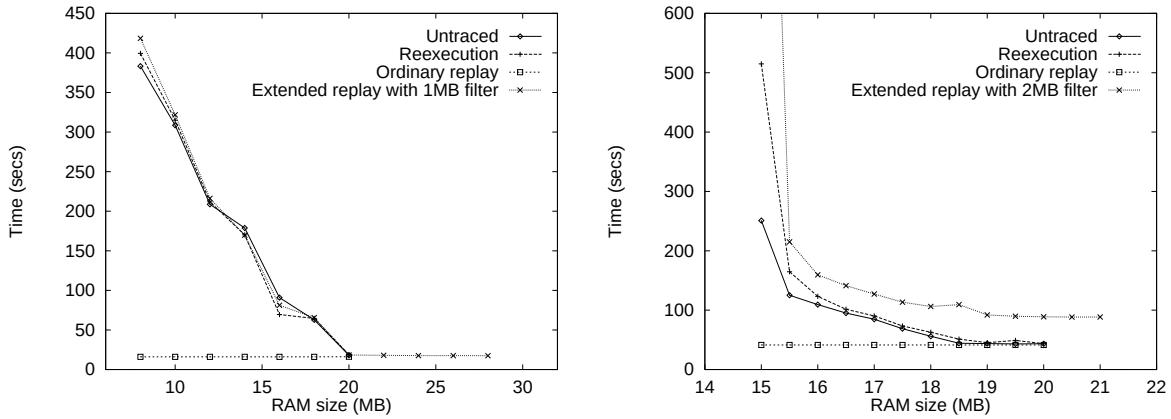
Table 4. Actual and predicted execution times for mSQL with files located on a remote volume.

The predictions by reexecution and extended replay are close to the actual times. This is not unexpected in the case of reexecution, as this method is better able to reproduce this component of the workload because the memory references are reproduced naturally as the original application code is reexecuted. The result for extended replay indicates that the deficiencies of ordinary replay have been overcome successfully.

7.2 Predicting effect of adding more memory

In these experiments, the machine was booted with varying amounts of RAM. Traces were captured of the applications executing with the minimum RAM size. These were then rerun on configurations with more memory. As before, in order allow the accuracy of ULTra's predictions to be quantified, the applications were also executed on each configuration with all tracing disabled. The application benchmarks we used for this experiment were `qsort` and C4.5, which are both memory intensive.

Figure 2 shows the results. As expected, trace reexecution successfully and consistently predicts the effect of increasing the amount of RAM. Ordinary replay fails spectacularly, since for these workloads the effect of adding RAM is to reduce paging. The results for extended replay are more encouraging, again showing that our approach is able to overcome the incompleteness seen with ordinary replay. The predictions for `qsort` are near perfect, but less so for C4.5. Extended ULTra successfully predicts the trend of increasing the size of RAM and correctly identifies the point at which the application no longer pages. However, the asymptotic behaviour indicates that the mechanism used for calculating user-level execution time is overestimating



(a) `qsort` (b) C4.5

Figure 2. Performance with varying RAM—actual and predicted

the time between events. Analysis of the traces showed that when summed, the inter-event execution times considerably exceeded the total execution time of the application. This discrepancy is caused by a small element of mis-attribution of user-level execution time to the application, which could be compensated to some extent by careful calibration.

8 Conclusions

System call tracing can be a very effective way of capturing an interactive workload. Trace-driven re-execution of the application can give very accurate predictions of application performance under varying configurations, but is not always applicable because of hard-to-reproduce phenomena. Trace replay is always applicable, but where paging is significant, has poor predictive value. We have presented a scheme which largely overcomes this problem.

The main remaining shortcoming of the approach is that for some applications – those with very poor locality – the trace capture overheads can be very large. This can be ameliorated to some extent by selecting an appropriate filter size.

Clearly, further work is needed to turn this work into an easy-to-use tool. Particular issues include reproducing synchronisation constraints in multi-process workloads, handling asynchronous signals, and handling events whose timing is determined externally.

References

[1] P. Ashton and J. Penny. A tool for visualizing the execution interactions on a loosely-coupled distributed system. *Software—Practice and Experience*, 25(10):1117–1140, October 1995.

[2] R. E. Barkley and C. F. Schimmel. A performance study of the UNIX system V fork system call using CASPER. *AT&T Tech. J.*, 67(5):100–109, 1988.

[3] 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.

[4] 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*, 26(3–4):261–278, April 2000.

[5] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of SHPCC'94*, May 1994.

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

[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. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software—Practice and Experience*, 20(12):1241–1258, December 1990.

[9] C. Merz and P. Murph. UCI repository of machine learning databases. `http://www.ics.uci.edu/~mlearn/-MLRepository.html`, 1998.

[10] A. Poon. Performance prediction using file-system traces and memory access patterns. Master's thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, United Kingdom, June 1997. MEng Final Report.

[11] J. R. Quinlan. *C4.5: Programs for Machine Learning*. The Morgan Kaufman series in Machine Learning. Morgan Kaufman Publishers, 1993.

[12] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, SE–3(1):94–101, January 1977.

[13] C. Turbyfill, C. Orji, and D. Bitton. AS^3AP: 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.