The following paper was originally published in the
Proceedings of the USENIX 2nd Symposium on
Operating Systems Design and Implementation
Seattle, Washington, October 1996

Studies of Windows NT Performance using Dynamic Execution Traces

Sharon E. Perl and Richard L. Sites
Digital Systems Research Center

Studies of Windows NT Performance using Dynamic Execution Traces

Sharon E. Perl and Richard L. Sites
Digital Systems Research Center
Palo Alto, CA
{perl,sites}@pa.dec.com

Abstract

We studied two aspects of the performance of Windows NT$^{\rm tm}$: processor bandwidth requirements for memory accesses in a uniprocessor system running commercial and benchmark applications, and locking behavior of a commercial database on a small-scale multiprocessor. Our studies are based on full dynamic execution traces of the systems, which include *all* instructions executed by the operating system and applications over periods of a few seconds (enough time to allow for significant computation). The traces were obtained on Alpha PCs, using a new software tool called PatchWrx that takes advantage of the Alpha architecture's PAL-code layer to implement efficient, comprehensive system tracing. Because the Alpha version of Windows NT uses substantially the same code base as other versions, and therefore executes nearly the same sequence of calls, basic blocks, and data structure accesses, we believe our conclusions are relevant for non-Alpha systems as well. This paper describes our performance studies and interesting aspects of PatchWrx.

We conclude from our studies that processor bandwidth can be a first-order bottleneck to achieving good performance. This is particularly apparent when studying commercial benchmarks. Operating system code and data structures contribute disproportionately to the memory access load. We also found that operating system software lock contention was a factor preventing the database benchmark from scaling up on the small multiprocessor, and that the cache coherence protocol employed by the machine introduced more cache interference than necessary.

1 Introduction

This work was triggered by two performance puzzles (circa 1995) related to Microsoft's SQL Server running on Alpha [SW95] PCs under the Windows NT operating system: how could we speed up the uniprocessor version, and how could we get closer to linear scaling for the multiprocessor version?

To answer these questions we found that we needed to look at the detailed behavior of the system under load. We created a tool for obtaining complete traces of the instruction and data streams executed by the processor in all operating system and application code. We proceeded to examine these traces and to use them to run simulations that revealed interesting properties of the then current system. The results of the simulations also had implications for future processor design.

The first puzzle we studied is how the processor bandwidth requirements of applications—the bandwidth required to service on-chip instruction and data cache misses from off-chip caches or memory—limits the achievable execution speed of the applications. This study was motivated by a discussion with some colleagues about using a prefetching strategy to improve performance. Their studies showed that prefetching would not help in the particular situation because there wasn't enough processor-chip pin bandwidth to support the workload. When pin bandwidth is a bottleneck, some common techniques for trying to improve performance do not help. These include multiple instruction issue, code scheduling, prefetching, and improved external cache latency. Pin bandwidth puts a ceiling on how fast an application can run. We discovered that pin bandwidth is indeed a bottleneck for interesting commercial applications, as well as for data accesses in one of the SPEC benchmarks that we studied.

The second puzzle we studied is how lock contention for a multiprocessor application limits the scalability of the application. Detailed execution traces reveal patterns of locking that may not have been expected by the operating system designers, application designers, or hardware designers responsible for the cache coherence protocol. Lock contention and the related cache coherence overhead prevent the application from scaling up beyond a small number of processors.

We choose these two aspects to study because they seem to be important factors in overall workload performance.

The contributions of this work are threefold: we provide evidence of processor pin bandwidth limitations and locking problems for commercial applications; we introduce a new tool for obtaining full traces of a system that allows us to study such problems; and we are making available some of the traces upon which our studies are based for other researchers to use.[1] Also, the understanding we gained from this work led to significant improvements to the hardware and software involved, so the results presented here do not fully apply to currently-shipping hardware and software.

In the next section we describe some of the more interesting aspects of the tracing tool, called PatchWrx. Section 3 describes our studies of pin bandwidth requirements for four different applications on two different Alpha processors. Section 4 describes studies of locking behavior of one of these applications—the Microsoft SQL Server database—on a small multiprocessor. Section 5 concludes.

2 PatchWrx

To understand the traces that are the input to our performance studies it is helpful to understand the properties of PatchWrx, the tool used to produce them. In this section we give an overview of PatchWrx, and then describe some of the highlights of its design and implementation.

2.1 Overview

PatchWrx is a software-only technique for capturing full time-stamped traces of the dynamic instruction stream (i-stream) and data stream (d-stream) of operating system and user code.

The goal of the PatchWrx effort is to capture traces of every single instruction executed throughout many seconds of a real operating system, running some complex workload on a non-microprogrammed multiprocessor. We wanted to build a software-only solution, rather than requiring one-of-a-kind hardware that cannot easily be applied at a customer computing environment. We wanted to gather traces with less than a factor of ten performance degradation, so that nothing in the operating system broke due to timeouts or excessive latencies. Finally, we wanted to work with arbitrary binaries for which we did not have access to the source code.

The technique we have adopted rewrites binary executable images, inserting patches that record in a log the target of every change of control flow (branch, call, return, system call, system return, interrupt, and return-from-interrupt), and some base register values for d-stream memory accesses. A reconstruction program working from the log and binary program images reproduces the trace of the full i-stream and d-stream that was executed. Our logs are typically about 5–10 times smaller than the resulting traces.

With the entire operating system patched for just branches (not load/store) and logging on, everything runs at about 1/4 of normal speed until the log buffer fills up. Then logging is turned off and the run speed is about 1/2 of normal speed. This is sufficiently fast that our personal machines have run patched all the time for over a year. With loads and stores patched in the operating system and applications, the worst slowdown we've seen is about 1/8 of normal speed. Patched images are 30-50% larger than the originals.

PatchWrx is an offshoot of the ATUM work in tracing using microcode [ASH86], and work with binary translation [SCK$^+$93]. Our approach is similar to other inline tracing efforts, but differs significantly in at least one dimension. Most published studies are of user code only [EKKL90, LB94], or are done on a single processor [BKW90, CB93], or require rebuilding source code [SJF92], or trace only cache misses, not all instructions [CHRG95, TGH92]. None use Windows NT. The excellent Shade paper [CK94] summarizes about thirty previous tools. Using that paper's classification, PatchWrx, like ATUM, traces executables, user and system code, multiple domains, multiple processors, signals, dynamic linking, and bugs, with performance similar to Shade.

We chose to produce traces rather than do on-the-fly data analysis [SE94] because of the difficulty of recreating complex execution environments months after the original investigation. With a detailed trace, questions asked a year or more later can still be investigated.

Somewhat like an electron microscope for computing, the PatchWrx approach is for studying a small amount of execution in excruciating detail, rather than summarizing long-running executions.

All of our experiments have been performed under Windows NT, version 3.5. The uniprocessor experiments were run on an Alpha AXP 150 with 128 MB of main memory. The multiprocessor experiments were run on a four-processor AlphaServer 2100, with 190 MHz processors and 256 MB of memory.

2.2 Trace Contents

The final output of PatchWrx is a trace containing the sequence of instructions executed by the operating system and all applications from the time logging was enabled

[1] Our uniprocessor traces are publicly available on CD-ROM to full-time faculty members. Contact one of the authors to obtain a copy.

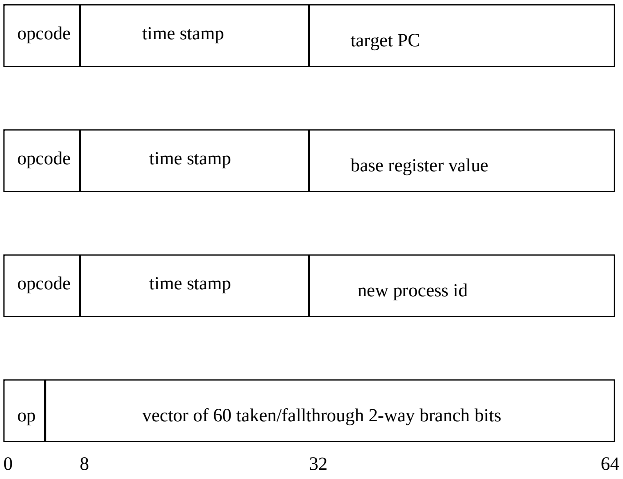

Figure 1: Formats of PatchWrx log entries. Each entry is 64 bits.

up until the log buffer (from which the trace is reconstructed) filled up.

Each instruction in a trace is tagged with its program counter value, and if it is a load or store instruction, the memory virtual address that is its source or target. In a multiprocessor trace, the instruction is also tagged with its processor number.

Some versions of our traces also contain timestamps on all branching instructions except for two-way branches. These are used to help line up the i-streams from different processors in the multiprocessor traces, and to compute lock holding times in our second study, described in Section 4.

2.3 Log Entries

While a patched system is running, a log is collected recording the information necessary to eventually reproduce a full system trace. A log is a sequence of entries describing branching events or data references. Figure 1 summarizes the different kinds of log entries.

As in most computer designs, the Alpha architecture has two basic forms of branching instructions: a jump to an address in a register, and a PC-relative two-way conditional branch. For each jump instruction, we record one log entry. For two-way branches, we accumulate a bit-vector recording the outcome of up to 60 two-way branches in a single log entry, one bit per branch. This gives us a compact encoding of the exact flow within a subroutine, taking only about 10% of the log entries.

When an interrupt or page fault occurs during logging, the address of the first instruction of the interrupt or fault handler is put in the log, along with the address of the first instruction *not* executed because of the interrupt (the instruction that would normally be executed immediately after the interrupt handler returns). This information is used during trace reconstruction to determine exactly where the interrupt occurred in the i-stream.

Additional log entries record information about memory load and store instructions, and process context switches.

In the Alpha implementation of PatchWrx, the log is recorded in a 45 MB portion of physical main memory that is reserved at boot time, and is therefore invisible to the operating system. The log buffer holds about 5.9 million eight-byte log entries, which is enough for 5–20 seconds of real time. There is so much information in a single reconstructed trace that we have not been motivated to try stitching multiple traces together [CHRG95, AH90]; a single reconstructed trace contains about 650 MB of dynamic i-stream with instruction and data addresses.

Recording the log in main memory is much faster than recording on disk or tape. Recording in physical memory instead of virtual memory allows us to trace the lowest levels of the operating system, including the page-fault handler, without generating recursive page faults. It also allows us to trace across multiple threads running in multiple address spaces, without needing to write a log entry to one address space while executing in a different address space.

2.4 PAL Subroutines for Logging

To implement the logging code, we use the Alpha architecture's PALcall instruction, which traps to one of a set of Privileged Architecture Library (PAL) specialized subroutines without disturbing any programmer-visible state, such as registers. These subroutines have access to physical main memory and to internal hardware registers and they run with interrupts turned off. We extended the PAL-code for Alpha NT with eight additional subroutines, and we modified some of the existing subroutines, as summarized in Table 1. Other architectures may have supervisor call or trap instructions that, in conjunction with modified operating system kernel interrupt routines, could be used to get a similar effect.

2.5 Collecting Data Addresses

It is possible to capture data addresses by patching all load and store instructions, but this fills up the log buffer quite quickly and so we would like to avoid it. We observe that many pieces of code use multiple references off the same base register. Since the i-stream reconstruction recovers the actual instructions executed, we will have these address specifications in the dynamic i-stream. Thus, for a sequence of references over which

PAL routine	Action/Recorded info.
PalReset	(Set aside log memory)
InterruptStackDispatch	next addr., interrupt target
SoftwareInterrupt	next addr., interrupt target
DispatchMmFault	next addr., page fault target
UNALIGNED	next addr., align. fault target
RFE	return from exception target
CALLSYS	sys. call target
RETSYS	return from sys. call target
SWPCTX	new process ID
pwrdent	read log entry from buffer
pwctrl	init. log, turn logging on/off
pwbsr	branch entry
pwjsr	jump/call/return entry
pwldst	load/store base register entry
pwbrt	cond. branch taken
pwbrf	cond. branch fall-through
pwpeek	(for debugging only)

Table 1: Logging-related PAL subroutines. First set are modifications to existing PAL subroutines. Second set (starting with pwrdent) are new PAL subroutines for PatchWrx.

changes to the base register value can be computed from the i-stream, we need only record the base register's value once. The effect is that only one out of every 5–10 load or store instructions is actually patched, and that for loops with constant strides through memory, only the initial base-register value outside the loop is traced.

Our patching and reconstruction algorithms are somewhat simplistic, causing the reconstruction to reach some loads and stores without knowing the base register value (as with certain interrupts). When this happens during reconstruction, we make up a random synthetic value for the base register, then track any incremental changes from there. Even in code with no load/store patches at all, this is surprisingly useful. The effect seen in the d-stream is that the bases of arrays and structures and stacks are random, but the relative access patterns within each aggregate are accurately reflected.

2.6 Handling Multiprocessors

When PatchWrx is running on multiple processors, the log entries for all processors are merged into a single log buffer. This allows us to see the dynamics of the interactions between processors. Writing a single merged log requires doing a multiprocessor-atomic update of the next-log-entry pointer, and requires encoding the processor number in each log entry.

Rather than interleaving single entries, we allocate chunks of 128 entries to each processor. This cuts down the frequency of atomic updates by two orders of magnitude and lets us encode the processor number once per chunk of entries, rather than in every entry. With log entries generated on each processor at the rate of about one per microsecond, a group of four chunks represents about 100 microseconds of real time on a four-processor system.

2.7 Trace Reconstruction from Logs

Reconstruction of a full trace given a log and a set of binaries is mostly straightforward. As described above, some special care and techniques are applied for obtaining data addresses from a limited set of base register values recorded in a log. Two additional tricky issues involve handling interrupts and merging multiprocessor traces.

The first issue concerns where to insert interrupts in a reconstructed i-stream. During reconstruction, one potential place to divert the i-stream to the interrupt handler is after all jump entries that precede the interrupt have been consumed, when the instruction that matches the not-executed address in the next log entry is encountered. If the not-executed instruction is inside a loop, the interrupt must be delivered during the right iteration. Loop iterations are controlled by jumps or conditional branches. So it is necessary to consume not only all the preceding jump entries in the trace, but also exactly the right number of conditional branch bits before delivering the interrupt. For this reason, we flush the partially accumulated taken/fallthrough vector into the log buffer just before recording the trace entries for an interrupt. This allows a perfect reconstruction of where to deliver an interrupt.

The second issue concerns merging traces from multiple processors. This requires special care because the time stamps within the entries come from four different cycle clocks (oscillators) on a four-processor system. These clocks are not synchronized with each other, and we observe drift of up to 100 parts per million (100 microseconds per second) within logs. All we really know about the clocks is that the first time stamp in chunk N was created after the first time stamp in chunk $N-1$ and before the first one in chunk $N+1$.

By carefully applying running inequalities between hundreds of chunks, we can map the drifting time bases into a single absolute time from the beginning of the log, within a window about two microseconds wide. We use this derived absolute time base to interleave the instructions in the reconstructed merged i-stream. The effect is that an instruction on one processor that stores to a shared variable is close (within dozens of instructions at a 4x tracing slowdown) in the final i-stream to an instruction on another processor that reads the shared variable and in fact sees the new value. Although we cannot guarantee that the store precedes the load in the final i-stream, we find this close enough for understanding multi-processor dynamics, including cache interactions.

2.8 I-stream Distortion

Because the patches introduce extra instructions and extra memory references on an instrumented system, there is necessarily some distortion in the timing of the reconstructed i-stream. There is little distortion of the i-stream itself, since the reconstruction excludes the patches. The paths through subroutines and the sequence of subroutine calls are the same in the reconstructed i-stream as they would have been in an uninstrumented system.

The reconstructed and uninstrumented i-streams differ in four subtle ways, however.

First, the patched images are bigger than the originals, so shared-library images loaded into consecutive memory locations end up at somewhat different addresses. This can have a slight effect on instruction cache (i-cache) simulations for large caches based on the traced instruction addresses. Patching expands images by multiples of 64 kilobytes, so the larger size images have no effect on small cache simulations; the low 16 bits of patched and original i-stream addresses are identical. There can be a similar slight effect on data cache (d-cache) and translation buffer simulations, as patches can also cause data to be moved.

Second, the patched images can take i-stream page faults on the pages containing the patches. While the patches themselves never show up in the reconstructed i-stream, the extra page fault traps and processing do. This appears to be only a slight distortion.

Third, timer interrupts happen four times more frequently in the traced code than in the uninstrumented code, due to the 4x tracing slowdown. In Windows NT 3.5, this increases the total number of instructions executed by about 1 percent. We have chosen to ignore this in our studies, but one could mechanically remove three of every four timer interrupts to further reduce the distortion.

Finally, external interrupts (disk, network, etc.) happen approximately four times sooner in the traced code than in the uninstrumented code (again due to the 4x tracing slowdown). To the extent that they are a consequence of the workload being traced, there are not more interrupts per million instructions executed—each one just occurs relatively sooner in the i-stream. Running an interrupt routine sooner than it would have run in the uninstrumented code can have a subtle effect on the memory access patterns of a processor, but there appears to be no interesting distortion when aggregated over several seconds.

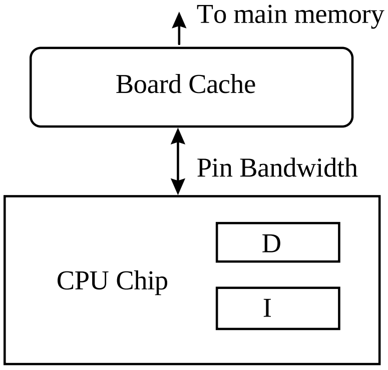

Figure 2: Processor pin bandwidth

3 Pin Bandwidth Study

Our first study using PatchWrx traces is a comparison of processor pin bandwidth requirements for a few different applications running under Windows NT on two Alpha processors. As shown in Figure 2, we are interested in the bandwidth between the processor chip (with on-chip i- and d-caches) and the board-level cache. This memory traffic is due to i- and d-cache misses. The question is: how many bytes per second need to cross the pins of a processor chip when running a given program at full speed? The answer is a function of the workload, including user and system activity, the properties of the processor chip, the clock speed, and the desired clock cycles per instruction (CPI). For our study, "full speed" is 1 CPI. The answers are interesting because, as we'll see, processor pin bandwidth is a first-order bottleneck for important applications.

3.1 Configuration

We looked at four applications:

1. Microsoft SQL Server, October 1994 beta version, a commercial database server, running the TPC-B [Gra91] benchmark. The trace contains 64.5 million instructions, spanning several hundred transactions.

2. GEM compiler [B+92] back-end, from Digital's commercially available optimizing C compiler, compiling a 3000 line C program. The trace contains 29 million instructions.

3. tomcatv, from the SPEC92 floating point benchmarks [SPE]. The trace contains 47 million instructions.

Application	#instr.	System Ld/st patched	Appl. Ld/st patched	Synthetic data addr.
SQL Server	64M	n	n	73%
GEM comp.	29M	n	Y	8%
Tomcatv	47M	n	Y	24%
Ear	83M	n	n	14%

Table 2: Trace characteristics.

4. ear, also from the SPEC92 benchmarks. The trace contains 83.84 million instructions.

We chose these applications because they give a variety of data points, from heavy memory system usage to light usage. At the time we started the experiments, the SPEC95 benchmarks were not yet available. It would be interesting to do the same experiments for the gcc SPEC95 benchmark, to see how it compares to the GEM results.

We took traces of these applications running on a uniprocessor Windows NT 3.5 system. The traces varied as to how much load/store patching was done. The reasons for not just doing full load/store patching relate to current PatchWrx limitations on image sizes and the slowdowns introduced by load/store patching. As shown in Table 2, none of the uniprocessor traces had loads and stores patched in the operating system. For the SQL Server and ear uniprocessor traces no loads or stores were patched in the application images. For the GEM compiler and tomcatv traces, load and store instructions were patched in the application images. None of the i-stream references in these traces are synthetic, but the d-stream synthetic references vary from 8% to 73%.

We used these traces to simulate the caching behavior of different processor chips under the assumption that the trace executes at 1 CPI. The simulation computes the number of bytes that would need to cross the pins to the board-level cache. These numbers were broken down by the source of the traffic: instruction reference misses, and data reference misses of a variety of kinds.

We looked at the behavior of each application on two different Alpha processors: the Alpha 21064 at 200 MHz, and the Alpha 21164 at 400 MHz.

The Alpha 21064 runs at 200 MHz and has 8 kilobytes each of direct-mapped write-through i-cache and d-cache with a 32-byte cache line size. Each instruction cache miss and data read (load) miss causes 32 bytes to cross the pins. Writes go into a group of four write buffers. A write to a different address causes 32 bytes to cross the pins to flush one of the dirty write buffers. The actual pin bandwidth on this processor is realistically about 300 MB/sec. The maximum bandwidth is about 600 MB/sec.

The Alpha 21164 runs at 400 MHz and has a single 96-kilobyte 3-way associative write-back combined i-cache and d-cache with a 64-byte cache line size. Each i-cache miss and load miss causes 64 bytes to cross the pins. This processor has a write-back cache: writes that hit in the cache do not cause any bytes to cross the pins but do make a cache line dirty. Writes and reads that cause a dirty cache line to be flushed ("victim writes") cause 16–64 bytes to cross the pins, depending on what portion of the line has been modified; dirty bits are kept for each 16-byte subblock. Writes that miss also cause a cache line to be allocated and filled from memory, so they cause 64 bytes to cross the pins for the read. Note that some of these reads will be unnecessary overhead if all 64 bytes are eventually overwritten ("redundant reads" as compared to "useful reads"). The realistic pin bandwidth on this machine is about 750 MB/sec, and the maximum bandwidth is about 1.6 GB/sec.

3.2 Results

For each of the applications on each of the processors we looked at a plot of the processor pin bandwidth requirements over time. This shows the dynamics of the traffic in addition to giving us upper bounds on the requirements.

3.2.1 SQL Server

Figure 3 shows the simulation results for pin bandwidth requirements on the Alpha 21064 and 21164 processors for the SQL Server workload. The x-axis marks time in chunks of 0.5 million instructions, and the y-axis shows the number of gigabytes per second of traffic across the pins for a given instruction trace.[2]

The left-hand graph shows the 21064 results. The traffic is broken down into i-cache misses (imiss), d-cache read misses (dmiss), and write buffer flushes (dwrite). At 1 CPI, the pins are overcommitted by a factor of two just on instruction cache misses alone. Three bytes of instruction miss in the cache for every four bytes executed. The required bandwidth for all traffic is about 1.2 GB/sec, or about four times the bandwidth that the processor realistically can deliver. In fact, the uninstrumented workload achieves about 4.3 CPI, giving us confidence that the cache simulations are valid. The simulation is intended to give lower bounds on CPIs based on pin bandwidth requirements; the extra 0.3 CPI in practice is due to non-zero latencies that are not included in the simulation.

[2] Note: the bandwidth pictures show only the first 25 million instructions of each trace; very little changes in any of the traces beyond that.

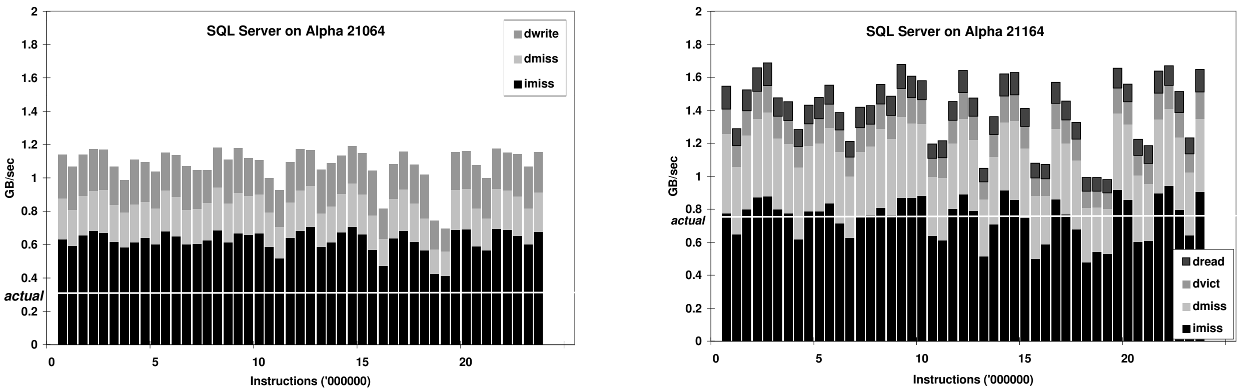

Figure 3: Pin bandwidth requirements of SQL Server running on Alpha 21064 and 21164 processors.

The right-hand graph shows the 21164 results. Here we split up the d-cache write traffic into victim writes (dvict), and useful and redundant reads triggered by write misses (dread). The required pin bandwidth here is as high as 1.7 GB/sec, so the pins are overcommitted by more than a factor of 2 at 1 CPI. The pins are still often overcommitted just on i-cache misses alone, although the i-cache miss traffic periodically dips down below the 750 MB/sec that the processor typically delivers.

3.2.2 Compiler Back-end

The second trace for which we simulated the pin traffic is from the GEM compiler run. The results are shown in Figure 4. The left-hand graph shows that the required pin bandwidth on the Alpha 21064 is about 600 MB/sec. This is about half as much as the SQL Server workload on this machine, with a somewhat lower ratio of i-cache to d-cache traffic. The pins are still overcommitted by a factor of two according to the simulation. The uninstrumented workload on this machine runs at about 2.5 CPI.

The right-hand graph shows the results for this workload on the Alpha 21164 400 MHz processor. Note that the i-stream here mostly fits into the 96-kilobyte cache, with about 200 MB per second of traffic left over. The total pin bandwidth requirements are generally under about 500 MB/sec, with occasional spikes over 800 MB/sec. Thus, pin bandwidth is mostly not a bottleneck for GEM on the 21164.

3.2.3 SPEC Benchmarks

For comparison with the commercial workloads, we chose two of the SPEC92 floating point benchmarks to examine.

Figure 5 shows the simulation results for the tomcatv trace. On the 21064, the i-stream essentially fits into the 8 kilobyte i-cache. The program repeatedly goes through phases where the pins are overcommitted by factors of 1.5 to 3 or more, and then are okay. The actual program runs at between 1 and 1.5 CPI on this machine. On the 21164, more of the workload falls within the bandwidth limitations, with the periodic lower plateaus well within the capacity of the processor pins, but there are still periodic benchmark array access bursts that are factors of two or more higher than the available bandwidth. Note that the instruction stream mostly fits in the cache, and so doesn't contribute much to the bandwidth requirements. Also, the demands for data bandwidth fall into a predictable pattern. Perhaps prefetching during times when bandwidth is underutilized could help for this workload.

Figure 6 shows the simulation results for the ear trace running on the same two processors. We see that even on the 21064, the pin bandwidth is essentially sufficient with required bandwidth about 200 MB/sec. The actual workload runs at 1 CPI or less (because of the dual instruction issue capabilities of the processor). The requirements for ear on the 21164 400 MHz processor are well within the range of the processor. A large part of the workload fits in the caches in this case.

3.3 Memory Maps

Looking at the patterns of memory accesses over time helps illuminate the results of the pin bandwidth simulations shown above. Figures 7–11 show the memory footprints for the first 25 million instructions in each trace[3]. The x-axis shows time from left to right, in millions of instructions executed. The y-axis shows virtual addresses accessed, modulo 4 MB (chosen because it is an interesting board cache size). There is a dot in the picture for each instruction address and for each load/store data address.

[3][except for the ear trace, which has about 21 million instructions shown in this version]

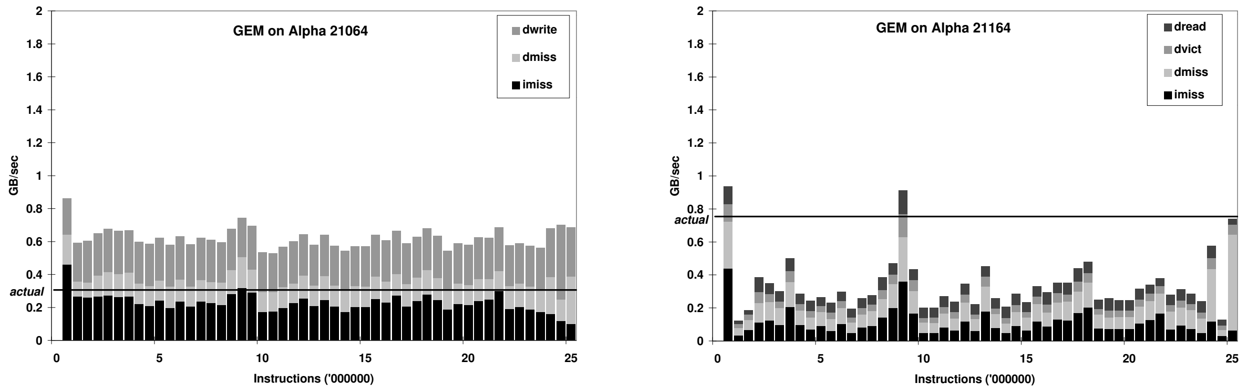

Figure 4: Pin bandwidth requirements of GEM compiler running on Alpha 21064 and 21164 processors.

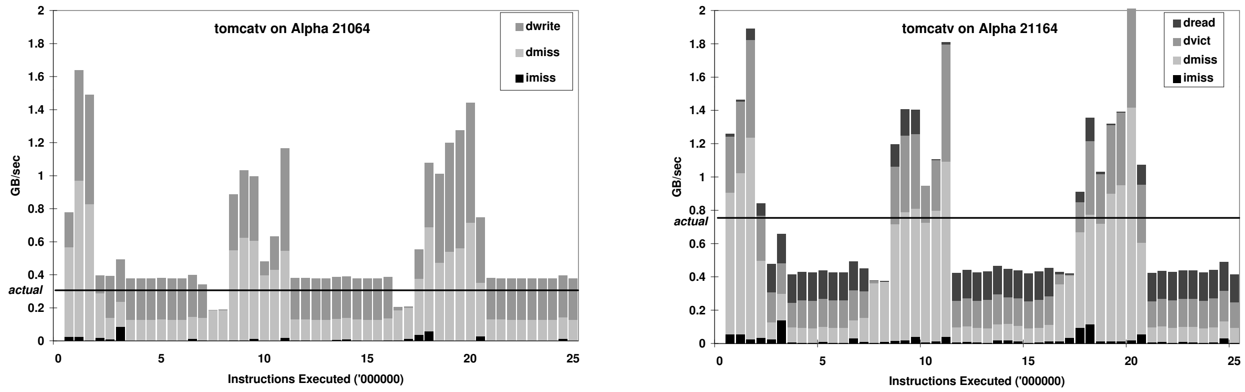

Figure 5: Pin bandwidth requirements of tomcatv running on Alpha 21064 and 21164 processors.

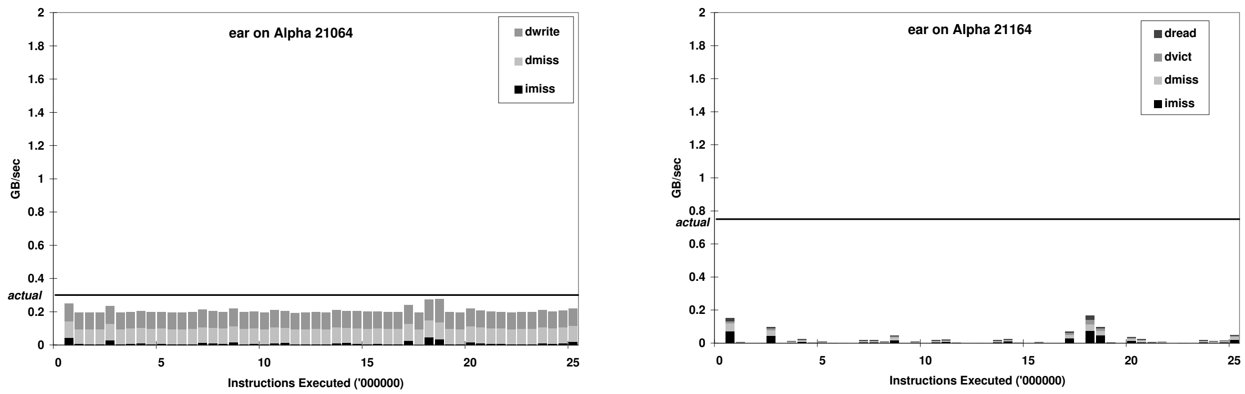

Figure 6: Pin bandwidth requirements of ear running on Alpha 21064 and 21164 processors.

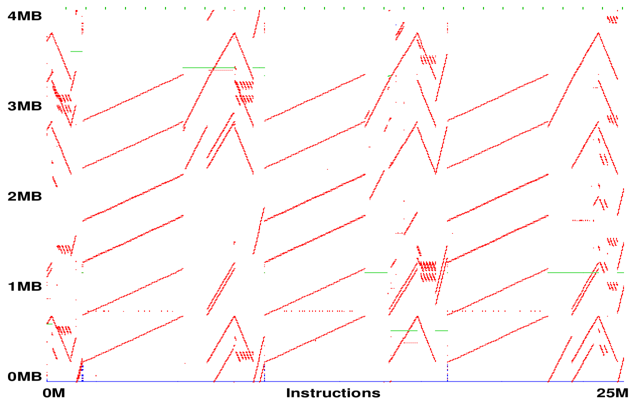

Figure 7: tomcatv memory access plot showing the distribution of user-space addresses over time. Looking at any vertical slice we see all the addresses (mod 4 MB) that were accessed within that time slice.

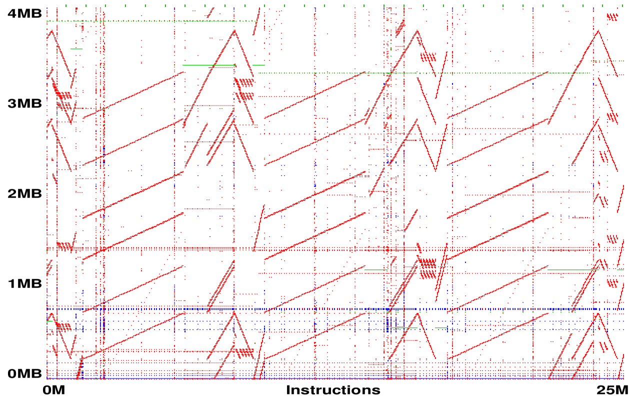

Figure 8: tomcatv memory accesses, user and system.

Figure 7 shows the memory footprint for the user-only part of the tomcatv benchmark. The bottom solid line shows the instruction fetches for the benchmark, all in a tight loop on a single page of code. The leftmost group of six upward-sloping lines are data references sweeping forward through six half-megabyte arrays. These are followed by some faster forward sweeps through different arrays, four backward sweeps, and the data pattern repeats. This program has very predictable branching patterns and very predictable data accesses.

In Figure 8, we have added the operating system part of the trace. The periodic vertical tics about one sixth of the way up from the bottom of the picture are the i-stream for the timer interrupt routine (about four times too often because of the tracing slowdown). The other dots that correlate with these are the data references made in the timer routine. The occasional vertical lines of high-density references are other operating-system activity, including the thread scheduler and network traffic. This operating system code has less predictable branching patterns and data accesses.

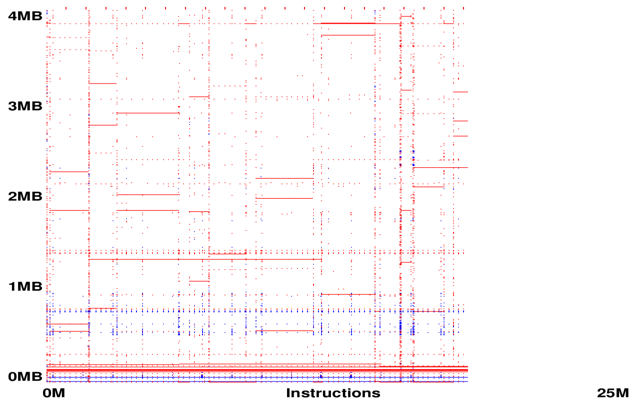

Figure 9: ear memory accesses, user and system.

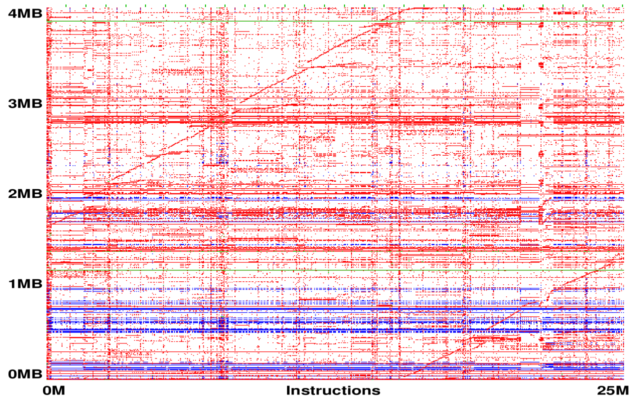

Figure 10: GEM memory accesses, user and system.

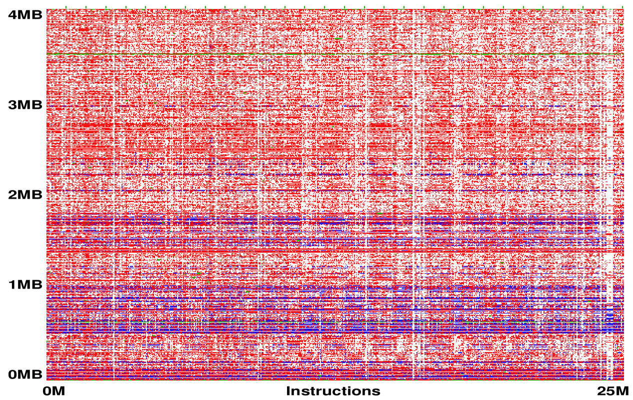

Figure 11: SQL Server memory accesses, user and system.

Figure 9 shows, on the same scale, a trace of both system and user addresses for the ear benchmark. The memory usage is relatively light, with few strong patterns other than the horizontal lines due to timer interrupts and some accesses to application data structures, and the vertical lines due to operating system activity.

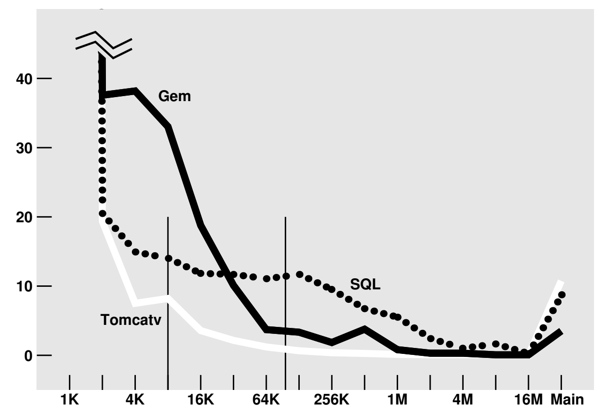

Figure 12: Incremental I-cache Hit Curves.

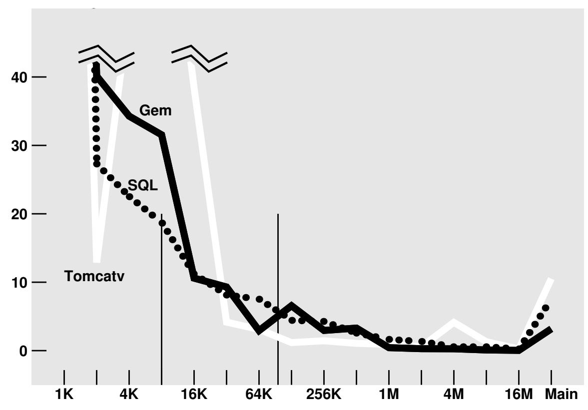

Figure 13: Incremental D-cache Hit Curves.

Figure 10 shows, on the same scale, a trace of the GEM compiler benchmark. The horizontal lines represent frequently used subroutines and frequently used data. The visually dominant diagonal line appears to be the buffer of program text being compiled; the compiler is sweeping forward through it (modulo 4 MB). The timer-interrupts are still there, but a bit harder to see. Most of the rest of the dots are data references to linked-list data structures used by the operating system. This workload has less predictable memory access patterns then tomcatv.

Finally, Figure 11 shows, on the same scale, a trace of the SQL Server benchmark. The trace looks somewhat similar to white noise—there is very little correlation between addresses. This application falsifies the premise of a cache: that accessing a location is a good predictor that the same or a nearby location will be accessed in the near future.

These memory maps highlight the differences between the memory requirements of the workloads. There seems to be at least some rough correlation between the density of the memory maps and the pin bandwidth requirements.

3.4 Cache Demand in More Detail

To understand the processor bandwidth results better, we looked at the *incremental cache hit* curves for cache sizes from 1KB to 16MB. Except for the smallest cache, the incremental hit rate for a given size cache is the number of *additional* hits that occur in the given cache and miss in smaller caches. Portions of the incremental hit curve for sizes bigger than processor on-chip caches represent misses that produce the off-chip bandwidth demand.

In this section, we used direct-mapped combined i- and d-stream caches and a line size of 64 bytes, and ran the simulations over the initial 25M instructions of each trace.

From the bandwidth charts above, we see a marked difference in i-stream miss rates across the uniprocessor workloads. Figure 12 shows the incremental i-cache hit curves for three of them (we exclude ear from this Section's discussion because its memory demands are uninteresting). For all three curves, the 1KB points are off the chart at 845-957 hits/1000 instructions. This is to be expected – if straightline code is executed with no repetition from 64-byte (16 instruction) cache lines, we would expect 15/16 hits, or 937 hits/1000 instructions. The short vertical bars mark 8K and 96K cache sizes.

The tomcatv i-stream fits almost entirely in 8KB, with a little tail that hits only in main memory. The GEM compiler i-stream extends out to 32KB before it drops off, and the SQL i-stream has significant incremental hits all the way out to a 1MB cache. This large footprint is why the SQL i-stream puts such a bandwidth load on the processor pins. This result is roughly consistent with the i-cache miss rates measured by [MDO94] for TPC-A and TPC-C (although they had a larger percentage of time in operating system code than we observed). The GEM footprint produces a large bandwidth load with an 8KB on-chip cache but a small load with a 96KB cache. The tomcatv footprint presents very little off-chip bandwidth load.

Figure 13 shows the incremental d-cache hit curves. All three d-streams show a tail of references that miss even in 16MB and go all the way to main memory. The tomcatv curve has peaks at 8KB (76 misses per 1000 instructions) and 4MB that correspond to sizes of frequently accessed arrays in that program. The 4MB peak is why the tomcatv d-stream puts a large periodic bandwidth load on the processor pins.

In Figure 14, we have broken down the SQL incremental i-cache hits by system and user code. The system hits are per 1000 system instructions, and the user hits per 1000 user instructions. The SQL system code is about 25% of the total i-stream. The system code not only has

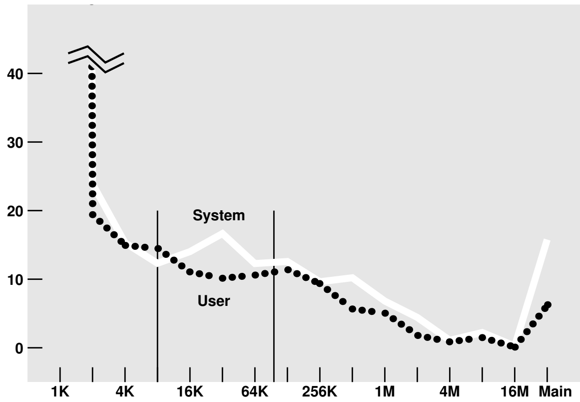

Figure 14: SQL User vs. System I-stream.

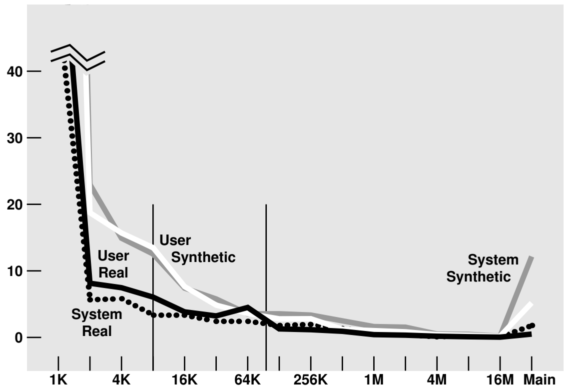

Figure 15: SQL User vs. System D-stream.

a bigger cache footprint, it has a larger tail of hits in main memory.

The SQL d-stream breakdown, Figure 15, shows system vs. user and also real vs. synthetic d-stream addresses. The d-stream has a high proportion of synthetic addresses, so is not very precise. The major difference between the real and synthetic addresses is a larger proportion of synthetic addresses that miss all the way out to main memory. For both the real and synthetic d-stream addresses, the system code has a larger footprint than the user code.

Both the GEM compiler and tomcatv i-stream breakdowns (not shown here) reveal the system i-stream to have a much larger footprint than the user i-stream. Similarly, the d-stream breakdowns show the system d-stream to have a much larger footprint than the user one (although tomcatv is close, due to the large array accesses).

In addition to observing larger footprints in system and SQL server code, the incremental cache hit curve allows us to compute the relationship between the CPI of a workload and the performance of various levels of the memory hierarchy. If we multiply each point of the incremental cache hit curves by the time it takes to service

	21064	21164
SQL	3.13	1.87
GEM	1.20	0.28
tom	0.04	0.06
ear	0.05	0.02

Table 3: Average bytes per instruction of pin traffic due to the i-streams of each of the benchmarks on the two processors studied.

that hit in a real memory subsystem, we get the total time spent waiting on memory accesses for that workload.

Using realistic times for early Alpha computer systems, the SQL i-stream and d-stream hits beyond 96KB each total about 1.8 CPI. Combined with a 100%-hits execution rate of about 1 CPI, these total about 4.6 CPI. This is roughly consistent with the measurements in [CB94] of 4.3 CPI, and 20-25% operating system time, and 39%+36% i-stream plus d-stream stall time, on the TP1 database workload on an older AlphaSystem 7000 computer.

3.5 Discussion

We conclude from these experiments that processor pin bandwidth is a bottleneck for some applications.

SQL Server needs about two to four times the bandwidth of tomcatv, and is limited to about 4 CPI on at least one current processor. According to another experiment that we ran, a future processor similar to the Alpha 21164 but running at 500 MHz would need at least 2 GB/sec of pin bandwidth to run the SQL Server benchmark at 1 CPI.

Both SQL Server and GEM have higher pin bandwidth requirements due to i-stream traffic than either of the SPEC benchmarks, as summarized in Table 3. This i-stream traffic contributes significantly to the pin bandwidth bottleneck for SQL Server. Therefore, chip designers should be looking at more than SPEC benchmarks if they want their future chips to run commercial workloads well.

Our experiments also suggest that it is increasingly important for algorithm designers to pay attention to memory structure and cache parameters if they want their code to run fast. One example of such an effort is the AlphaSort work [NBC+94], where careful attention to these memory details paid off handsomely.

4 Multiprocessor Study

Our second study examines the locking behavior of SQL Server running the TPC-B benchmark on a four-processor AlphaServer 2100 symmetric multiprocessor.

The traces were taken for SQL Server version 6.00.85a (March 1995 beta release), running Windows NT 3.5, with 12 clients running transactions against the database. We chose this number of clients to minimize the idle time of the system, which ended up being about 2–3%.

The trace contains about 69 million instructions. In contrast to the uniprocessor traces, operating system loads and stores are patched. For the application code, only loads and stores to lock variables are patched.

We use the trace in two ways. First, we look at a picture showing the timeline of software lock acquisition and release activity. Then, we simulate the cache coherence protocol employed on the machine to understand how the locking activity affects communication among the processors.

4.1 Software Lock Activity

Figure 16 shows the timeline of locking activity for a 20-millisecond portion of the SQL Server trace, encompassing portions of several transactions on the four processors. The picture shows groups of four lines (with backgrounds alternately shaded white and light gray), one line per processor, with shaded bars representing the elapsed times that locks are held. Time runs from left to right and top to bottom. Each group of four horizontal lines represents 2 milliseconds of elapsed time.

In this picture, we distinguish three cases: the thick black bars are the operating system's dispatcher lock (KiDispatcherLock), the thick gray bars are all other lock holding times, and the thin black lines are times spent spinning while waiting to acquire a lock.[4] In the rectangle at 0.030 seconds, for example, processor 0 starts out holding no locks, processor 1 holds the dispatcher lock, processor 2 is spinning on the dispatcher lock, and processor 3 hold no locks. Processor 1 then releases the dispatcher lock and processor 2 acquires it. Four other locks are used, then processors 1 and 2 use the dispatcher lock again. Finally, processor 1 uses another lock and processor 0 uses a lock.

Lock acquisitions and releases are not reported directly in the trace, but rather, must be detected by pattern matching against the common sequences of instructions used for this purpose. Therefore, the picture showing lock acquisition and release points is not perfect. However, we have taken some care to ensure that the picture is reasonably self-consistent, and consistent with the code.

While this picture shows just a small slice of the trace, it reveals several interesting behaviors. The dispatcher lock is held for relatively long stretches of time (200–900 instructions). Also, it is held for a large percentage of the time—about 45% of the total elapsed time. This suggests that the workload could not scale up beyond eight processors, which would cause this lock to be held nearly 100% of the time.

More that 16% of the time is spent spinning while waiting for a lock, usually the dispatcher lock. This spin time would go up substantially as the lock-held time approaches 100%. Note the convoy effect in the second rectangle, at about 0.041 seconds. Processors line up spinning for the dispatcher lock.

During the long stretch where processor 0 holds the dispatcher lock, a disk interrupt arrives and the processor services it while still holding the lock. Other processors are waiting to get the lock during that time.

All of the evidence above suggests that the dispatcher lock is a bottleneck. Because of the convoy effect, it would be nearly impossible to get beyond about 6 processors' worth of work from this code.

The picture reveals that in addition to being held for a long time overall, the dispatcher lock is held frequently. This is because the SQL Server user code goes through the dispatcher lock to block when one of its own locks is unavailable. This type of lock usage was perhaps not anticipated by the Windows NT implementers. Showing the picture to the Windows NT kernel and SQL Server implementers at the same time brought out assumptions that each group was making about the other's code.

The full-color version of this picture makes other locking patterns visible. For example, there is a repeated pattern of nine lock acquisitions (some of which are the same lock repeatedly acquired and released) in the same order when manipulating the operating system page tables. Examining the code may suggest ways to make the associated operation more efficient.

4.2 Interference from Cache Coherence

The software lock activity above for achieving mutual exclusion is best tuned by changing the software. We also looked at multiprocessor hardware cache interference caused by writes on one processor affecting caches on other processors.

We looked in detail at the behavior of the cache coherence protocol with four AlphaServer 2100 processors. A pure invalidate protocol is used—a write by one processor to a cache line invalidates all other cached copies of the line in other processors. We found that 40% of the board-level cache misses in the 4-processor SQL trace were due to interference from other processors—invalidates due to writes of shared variables.

Most of these misses were in fact to about 24 cache lines containing software locks. (Note that the time lost on servicing these misses is smaller than the time lost above spinning on held locks.) Some of the invalidations

[4] We usually look at this picture in color, with different colors and line widths for different locks, making more of the locking behavior apparent at once.

Figure 16: SQL Server lock activity. Time runs left-to-right and top-to-bottom.

came from false sharing. The programmers who wrote the code assumed 32-byte cache lines and took care to put locks in different 32-byte lines than other variables. However, this particular machine has 64-byte lines, so a lock and another unrelated shared variable could end up in the same line and cause unnecessary invalidations. The ability to track shared sub-blocks in the hardware would help this problem. A better cache coherence protocol for this workload would be a hardware design that writes through for shared blocks, having recipients update or invalidate depending on how recently they had touched the line. Such a design was used in the Alpha Demonstration Unit [TCS93].

4.3 Discussion

The Windows NT 3.5 operating system code holds the dispatcher lock for so long because it is doing a full context switch while holding the lock. It would be better to make the decision about what to dispatch next while holding the lock, release the lock, then perform the actual context switch without delaying the other processors. Allowing interrupts while holding the dispatcher lock also lengthens the path and slows down other processors. It would be better to turn off interrupts before acquiring the lock and turn them back on after releasing the lock.

The application code that triggers too-frequent use of the dispatcher lock could adopt other locking strategies, such as trying a handful of times to acquire an application lock before blocking or to do user-level dispatching between multiple transactions.

Examining the dynamic locking behavior of the system reveals properties that are more difficult or impossible to see from statistical information. We can see clearly the convoy effect for locks that are in high demand, and we can see patterns of locking that may suggest inefficiencies in the code. We can also identify cases where locks are probably not a performance problem, as is true for most of the locks in our trace.

Having the traces available for simulating cache coherence protocols helped us to understand the causes of scaling problems in our multiprocessor system, and can help the designers of future systems to avoid these problems.

5 Conclusions

These studies have demonstrated that full, dynamic execution traces are useful for designing future systems, as well as for diagnosing difficult performance problems of current systems.

Obtaining such traces is practical using PatchWrx. The overhead is low enough on the uniprocessor systems (about a factor of 2 slowdown) that we always ran the systems with instrumentation in place. Overhead on multiprocessor systems was reasonable enough to get useful traces. PatchWrx traces are in the range of about 50–100 million instructions, which was enough to observe interesting dynamic behavior in the system while still being practical to collect.

The specific versions of software that we studied are inevitably already out of date with respect to currently available versions, and some of the specific problems revealed in our studies may have been fixed by now. In fact, the Microsoft SQL Server 95 and Windows NT 3.51 and 4.0 products contain performance improvements [5] that resulted from our studies [Gra95] . However, the same techniques and tools can be used to look for performance problems in the new versions and the traces still provide valuable input for testing future chip designs.

We find that real pieces of large software are bigger than any practical on-chip caches, and therefore use much more processor pin bandwidth. Pin bandwidth can be a bottleneck to running commercial workloads at the desired speed. For the near future at least, computer design is memory design, and software design is memory design. Pin bandwidth is an important consideration (as much as memory latency) because it sets a lower bound on the number of clock cycles per instruction needed to execute a workload.

Although this paper emphasizes the effect of bandwidth on the performance of these processors, one shouldn't ignore the effect of latency. If there is insufficient bandwidth to perform a given workload, the workload will run proportionally slowly, and there is nothing software can do to compensate for this. Systems that have sufficient bandwidth but high memory-access latency can possibly perform well, but only for software that is designed to hide the high latency. Both high bandwidth and low latency are necessary for peak performance on most commercial software.

6 Acknowledgments

A number of people at Digital have contributed to this effort. Benn Schreiber, Tom van Baak, Miche Baker-Harvey, and Joe Notarangelo gave us immeasurable help in getting the NT PAL-code modifications running. Rich Witek wrote the original single-processor boot PAL-code. Mike Burrows contributed expertise on binary modification techniques. Wim Colgate provided the key insight into booting NT multiprocessor systems. Dave Hunter generously provided time on a four-processor AlphaServer 2100 system. Chuck Thacker, Dave Conroy, Bill Weihl, Greg Nelson, Amitabh Srivastava and Alan Eustace provided continuing encouragement. Our summer interns, Cliff Mercer and David Martin struggled with early software and each provided key insights for increasing performance of the workloads studied. David Martin was responsible for the cache coherence protocol simulations. We also thank Jim Gray and David Cutler at Microsoft.

The observations of Richard Draves and the anonymous OSDI referees substantially improved the paper.

[5] Improvements include: some collapsing of the 36 memory allocation calls per TPC-B transaction, each of which calls EnterCriticalSection; replacing a slow locking sequence macro expansion with a more efficient one, worth 5-8% all by itself; removing unnecessary calls to floating point initialization routines; avoiding calls to OS services for CPU time and clock time; compiler improvements in setjmp/longjmp implementation; minimizing context switches; and increasing cache locality.

References

[AH90] Anant Agarwal and M. Huffman. Blocking: Exploiting spatial locality for trace compaction. *Performance Evaluation Review*, 18(1):48–57, May 1990.

[ASH86] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proc. of the 13th International Symposium on Computer Architecture*, pages 119–127, June 1986.

[B+92] David S. Blickstein et al. The GEM optimizing compiler system. *Digital Technical Journal*, 4(4), 1992. Also available as http://ftp.digital.com/pub/Digital-/DECinfo/DTJ/axp-gem.ps.

[BKW90] Anita K. Borg, R. E. Kessler, and David W. Wall. Generation and analysis of very long address traces. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 270–279, May 1990.

[CB93] Brad J. Chen and B. Bershad. The impact of operating system structure on memory system performance. In *Proc. of the 16th International Symposium on Operating Systems Principles*, pages 120–133, December 1993.

[CB94] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of Alpha AXP performance using TP and SPEC workloads. In *Proc. of the 21st Annual Symposium on Computer Architecture*, pages 60–70, April 1994.

[CHRG95] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *ACM SIGMETRICS*, pages 1–13, May 1995.

[CK94] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS*, pages 128–137, May 1994.

[EKKL90] Susan J. Eggers, David Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *ACM SIGMETRICS*, pages 37–47, May 1990.

[Gra91] Jim Gray, editor. *The Benchmark Handbook*, chapter 2, pages 79–117. Morgan Kaufmann, San Mateo, California, 1991.

[Gra95] Jim Gray, December 1995. Private communication.

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software—Practice and Experience*, 24(2):197–218, February 1994.

[MDO94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–155, October 1994.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A RISC machine sort. In *SIGMOD Conference*, pages 233–242, 1994.

[SCK+93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 1–1, 1994.

[SJF92] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs. Address tracing of parallel systems via TRAPEDS. *Microprocessors and Microsystems*, 16(5):249–261, 1992.

[SPE] SPEC CPU92 benchmarks. World Wide Web URL: http://www.specbench.org/osg-/cpu92/.

[SW95] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, Newton MA, 2nd edition edition, 1995.

[TCS93] Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart. The Alpha Demonstration Unit: A high-performance multiprocessor. *Communications of the ACM*, 36(2):55–67, February 1993.

[TGH92] J. Torrellas, A. Gupta, and John Hennessy. Characterizing the cache performance and synchronization behavior of a multiprocessor operating system. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.