Monster:

A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures

---

University of Michigan Tech Report
May 6, 1992

David Nagle
Richard Uhlig
Trevor Mudge

Abstract

To enable computer designers to better evaluate the architectural needs of operating systems, we have developed *Monster*, a tool which combines hardware and software monitoring techniques to unobtrusively obtain system performance data. This report is split into two major parts. In Part I, we argue the need for OS performance evaluation tools, summarize previous hardware and software based monitoring techniques, discuss our design of Monster and finally present an analysis of compilation workloads which test and demonstrate Monster's capabilities. In Part II, we detail our plans for future studies in which Monster plays a central role.

Part I: Monster

1 Introduction

In recent years, a number of architectural and operating system trends have become evident. In architecture, semiconductor advances have allowed processor SPECmarks to double every 18 months [SPEC 91]. Coupling this with denser integration levels, designers have been able to change and better integrate register files, pipelines, memory management hardware, floating point hardware and cache organizations in ways that previous semiconductor technologies would not allow [Hennessy and Jouppi 91].

In operating systems, new technologies and demands on system software have also forced a number of changes. For example, high speed networks have enabled distributed computing while multiprocessors are bringing parallel programming into the computing mainstream. Further, rapid computer design cycles have increased the need for more portable and modular software systems, resulting in a movement toward microkernel structured operating systems such as Mach [Golub et al. 90].

These trends have caused a considerable change in the way that operating systems and computer hardware interact which in turn has created a need to reconsider and evaluate these interactions. In their paper entitled "The Interaction of Architecture and Operating System Design," [Anderson et al. 91] discuss some of these effects:

- **Interprocess Communication (IPC).** The decomposition of large, monolithic kernel operating systems into a microkernel and multiple, user-level server tasks which communicate across address spaces is making good IPC performance vital. This performance is mostly dependent on how efficiently an architecture supports system calls and interrupts which, in turn, depends on how quickly processor state can be saved and the appropriate handlers can be invoked. With IPC performed via remote procedure calls (RPC), a significant portion of time can also be spent performing block memory operations for checksum computation and parameter marshaling [Schroeder & Burrows 90].
- **Virtual Memory (VM).** Microkernel operating systems are increasing the number of address spaces that reside in a machine. This, combined with increased switching between address spaces can stress hardware resources such as address translation buffers (TLBs) and page tables.
- **Thread Management.** The degree to which future multiprocessors can exploit fine grained parallelism depends on how inexpensive lightweight thread context switching will be. This expense is a direct function of how quickly process state, in the form of CPU registers, can be saved to and restored from the memory system.

In summary, Anderson et al. argue that the frequency and duration of time that hardware spends performing fundamental OS operations is changing. Furthermore, while current architectural features have done much to improve SPECmark performance, these same structures are providing little help to applications that are more reliant on operating system resources. For example, the multistage instruction pipelines and larger register sets of new microprocessors can adversely affect the speed and complexity of trap and interrupt handling. Also, while on-chip data caches can boost performance once a working set becomes cache resident, they are of little utility when data regularly comes from or goes to uncached I/O buffers as is often the case with block memory operations. Finally, the floating point unit hardware which has done so much to boost SPECmarks in newer systems is of little assistance to the operating system.

Computer designers need to begin considering architectural features which better support the operating system. But to make sensible choices, they need tools that are capable of viewing both user and kernel modes of execution and which cause minimal disturbance to the system under analysis. This paper seeks to address these issues by presenting *Monster*, a new tool which combines hardware and software monitoring techniques to enable the analysis of systems built from new computer architectures and operating systems.

The rest of Part I is organized as follows: Section 2 summarizes previous work on hardware and software based analysis tools. Section 3 discusses issues regarding the design of Monster. Section 4 demonstrates some of Monster's capabilities on a test analysis of compilation workloads. Finally, Section 5 summarizes major observations from the test analysis.

2 Previous Work

Previous work in performance analysis falls into two basic categories: *software-based* and *hardware-based*. Both of these approaches offers advantages and disadvantages. For example, software techniques provide flexibility by being easy to port to new machines and by not requiring any special equipment. However, software techniques cannot make fine-grained[1] measurements without perturbing the system. Conversely, hardware techniques are more passive and provide access to finer-grained events not visible to software, but often require special equipment and are tied to a specific machine architecture. The following two sections summarize some common software and hardware analysis techniques.

2.1 Software Monitoring Techniques

There are two basic software techniques for performance analysis: *single-step tracing* and *execution sampling*.

1. "Fine-grained measurements" involve the monitoring of events that can change as frequently as once every machine cycle.

The UNIX `ptrace()` facility is an example of single-step tracing. `ptrace()` allows a parent process to collect information about the execution of a child process by stepping through it's execution one instruction at a time. However, because the traced process must pass through the operating system on every instruction (when making fine-grained measurements), the slow-down in performance makes it difficult to collect traces that span more than a few seconds of execution time.

To overcome this speed limitation, several code annotation systems such as `AE`, `pixie` and `IDtrace` have been developed [Larus 90, MIPS 88, Pierce 92]. These tools embed extra code directly into an executable image so that when a program is run, the extra code will output data describing the program's behavior. This data can be processed to provide execution statistics and instruction or memory address traces. The trace data can either be output to a file, piped directly into a simulator or stored in a special RAM trace buffer [Mogul and Borg 91]. By processing the trace on-the-fly, it is possible to analyze billions of trace addresses or instructions without the need for disk storage [Mogul and Borg 91, Olukotun et al. 91].

Execution sampling is an alternative software technique in which machine state (program counter, execution mode, etc.) is sampled at regular intervals. Usually the sampling rate is determined by the clock interrupt frequency. There are a number of UNIX facilities that are based on this technique. For example, the `time()` system call provides a rough estimate of how much time a program spends in user and kernel modes. Similarly, `iostat()` and `vmstat()` provide information on CPU idle time and paging performance, respectively.

In general, software-based techniques are useful for locating gross system bottlenecks. However, because software monitoring distorts the original program, it does not work well for collecting traces of the operating system or for collecting fine-grained OS data and hardware events. To obtain this information, it is sometimes necessary to turn to hardware-based monitoring techniques.

2.2 Hardware Monitoring Techniques

Most hardware monitors fall into one of two categories [Agarwal 89].

- **Event counting monitors:** The monitor triggers on specific events and then increments a counter.
- **Event tracing monitors:** The monitor captures interesting events and stores them in a trace buffer for post processing.

These two different approaches have various advantages and disadvantages. Typically, event counters can monitor for virtually indefinite periods of time without having to stall the system under analysis. This compares favorably to tracing monitors that use memory buffers which may fill within a microsecond. The monitor then must resort either to sampling the complete execution or to stalling the system while the trace buffer is emptied.

Trace buffers provide the advantage that they can be post processed, whereas event counting must be performed in real time[1]. Hence, tracing allows more sophisticated data analysis without the cost of elaborate hardware triggering facilities.

One of the best known event counting monitors is the "micro-PC monitor" [Emer and Clark 84] built for the VAX-11/780. This monitor's registers formed a histogram of how many times each microinstruction executed. From this histogram Emer and Clark were able to compute instruction distributions, memory system stalls, some statistics on user code behavior and the average number of micro cycles per VAX instruction.

An example of the event tracing approach is Agarwal's modifications to the VAX 8200 microcode [Agarwal 89]. Here, microcode was added to produce traces of user and kernel memory references. Agarwal's results showed the VAX's traces contained up to 50% system references. These system references can double the cache miss rate when compared against traces with only user references [Agarwal 89]. This is one of the first studies to show that operating system references can substantially impact overall performance.

There are also several examples of hardware monitoring facilities designed directly into a computer's architecture. The CRAY X-MP has four groups of performance counters that can measure a variety of machine functions [Nelson 89]. Also, the IBM RS/6000 [Groves and Oehler 91] and the DEC Alpha [DEC 92] provide high resolution timers that can measure different aspects of the system's performance.

3 Monster

Monster's primary purpose is to serve as a tool that will enable us to examine the interaction between operating systems and computer architectures. The design of Monster is a hybrid approach that attempts to combine the strengths of traditional hardware and software based techniques while avoiding their weaknesses.

The hardware portion of Monster consists of DECstation 3100 physically connected to a Tektronix DAS 9200 logic analyzer. The DECstation hardware has been modified to make its R2000 CPU pins accessible to the logic analyzer. The R2000 cache is off-chip, so the logic analyzer probes sit *between* the processor and cache. Because the probes have access to every address, data, instruction and control signal emanating from the CPU, it is possible for Monster to detect a wide variety of hardware events. Monster's hardware is shown in Figure 1.

The software portion of Monster consists of the DECstation operating system, Ultrix, which has been annotated with markers to indicate the entry and exit points of different regions of code. These software markers assist the hardware portion of Monster by mak-

1. By *real time* we mean that the processing must be done at the same speed and concurrently with the system being monitored.

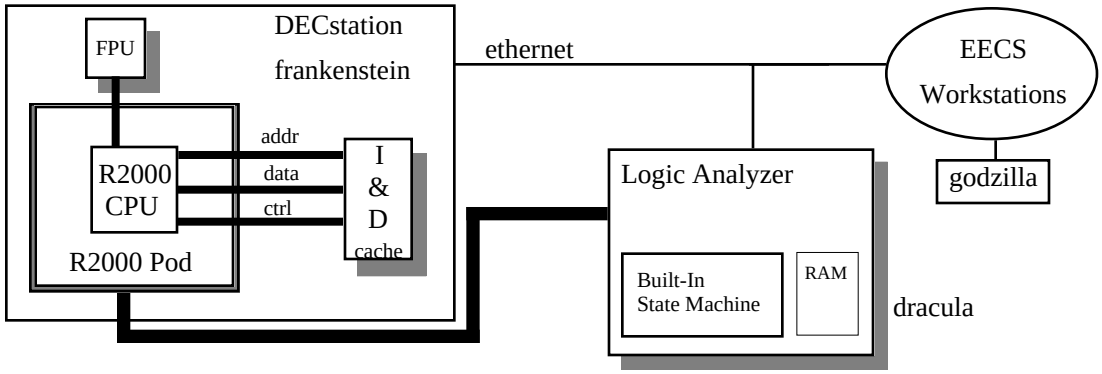

Figure 1: Monster Hardware

ing it easier to detect code boundaries so that monitoring can be turned on and off depending on the current region of execution.

3.1 Event Tracing with Monster

Monster's logic analyzer contains a high speed 512K RAM which can be used as a trace buffer. By using the logic analyzer's triggering capability, it is possible to filter incoming signals in real time and selectively capture only specific events. Tracing can also be used to measure the precise time between events by writing both a time stamp and the event to the trace buffer.

3.2 Event Counting with Monster

The logic analyzer component of Monster also contains a programmable hardware state machine. The main building blocks of this state machine hardware are eight 96-bit pattern recognizers, two 32-bit high speed counters and 16 states. A state machine specification defines transitions between states and in which states the high speed counters should be incremented. The state transitions depend primarily on whether or not the bit pattern recognizers match certain logical values detected at the pins of the DECstation CPU.

By *carefully* programming this state machine hardware, it is possible to implement event counting with Monster. But because event counting must be performed in real time several complications can arise. In particular, with the R2000 processor, pipeline flushes and cache misses can occur in any cycle, making it difficult for the monitor to know if fetched instructions actually complete. The next section will discuss our solutions to these and other monitoring problems in greater detail.

3.3 Monster's Hardware Component

The most basic monitoring experiment is to count occurrences of a specific opcode. The naive solution is to immediately increment a counter in the state machine every time the opcode is detected by a bit pattern recognizer. In pseudo-code, here is an example state machine which implements this experiment by immediately counting all occurrences of the opcode `add`:

```
if (instruction_bus == add)
   then counter++;
```

Unfortunately, this counts many more `add` opcodes than are actually executed. Why? The problem stems from how the R2000 handles instruction cache misses. A miss is detected one cycle *after* the opcode has been fetched from memory and is read into the processor. From Monster's perspective, cache misses and cache hits look the same during the fetch cycle. Therefore, if only the fetch cycle is monitored, every miss that outputs an invalid `add` (a stale `add` opcode that is still in the cache), will get counted as a valid `add`. We call these *phantom opcodes* because the monitor sees them enter the processor, but they are not executed.

To overcome this problem, Monster must keep track of both the current and previous cycles and only count an opcode that is seen in the first cycle if a miss signal does not occur during the next cycle. We call this a *one-cycle-history* state machine and is shown below:

```
state1: if (instruction_bus == add)
           goto state 2:
        else
           goto state 1;
state2: if (miss_signal!= miss)
           counter++;
        also  if (instruction_bus == add)
                 goto state 2:
              else
                 goto state 1;
```

The "`if (miss_signal != miss)`" and the "`also if (instruction_bus == add)`" are processed in parallel to handle the possibility of back to back `add` opcodes.

Pipeline flushes due to hardware interrupts can also affect the accuracy of event triggering. Upon return from an exception, instructions that are flushed will be refetched. A simple state machine will count these opcodes twice, once before the flush and once again during the refetch. To overcome this problem, Monster interprets the R2000 control signals to detect pipeline flushes and takes corrective action. Basically, this was implemented by extending the one-cycle-history state machine to a four-cycle-history state machine.

Many of our experiments only monitor within a specific region of execution. To assist the monitoring hardware, we surround regions of interest with special "marker" opcodes. These markers were formed from `nop` instructions and embedded in the OS to mark kernel entry, kernel exit and other interesting regions of code. To simplify detection of the markers by a Monster state machine, they were placed in uncached memory with interrupts turned off. This technique guarantees that markers will not be flushed from the pipeline or appear as phantom opcodes. Markers are one way that Monster's software and hardware work together to simplify monitoring. The next section discusses some of the other ways that Monster's software component assists it's hardware component.

3.4 Monster's Software Component

Another problem with monitoring is the handling of interrupts and exceptions. For example, to monitor a region of code, markers are placed at the region's entry and exit points so the Monster hardware knows when to start and stop monitoring. However, most code can be interrupted, forcing the machine to re-enter the kernel and handle the interrupt. Since it may be undesirable to monitor the interrupt handler, one quick solution is to turn off interrupts in the region of code being monitored. This, however, could seriously distort the behavior of the OS. Further, while interrupts can be masked, exceptions cannot. Therefore, turning off interrupts does not provide a complete solution.

To solve this problem, several pieces of code were added to the kernel. First, a marker was placed at the kernel entry so that when an interrupt or exception occurs within the region of code being monitored, the kernel will reenter itself and emit the entry marker. The Monster hardware looks for this entry marker and stops monitoring until the kernel returns to the region of code being analyzed. However, because the kernel may not return directly back to that region of code, Monster cannot resume monitoring when it sees any kernel exit. Therefore, a small piece of address check code was inserted into the kernel `exit()` routine. This code checks the return address to see if it falls within the bounds of the routine being monitored. If it does, then another special marker is emitted to alert Monster to resume monitoring. This scheme is depicted in Figure 2

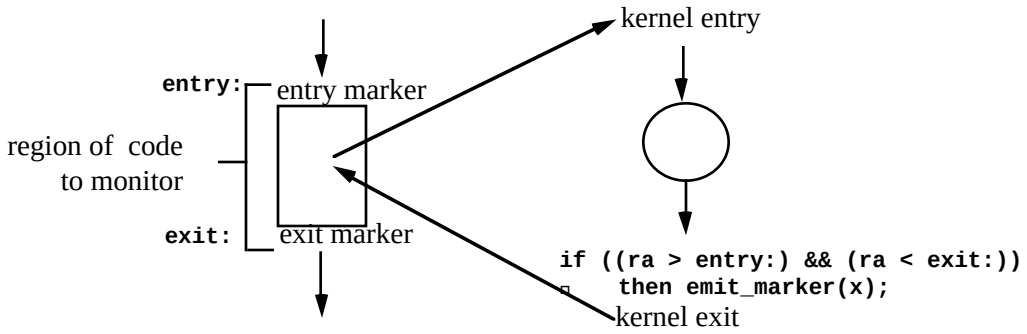

Figure 2: Monitoring Regions of Code

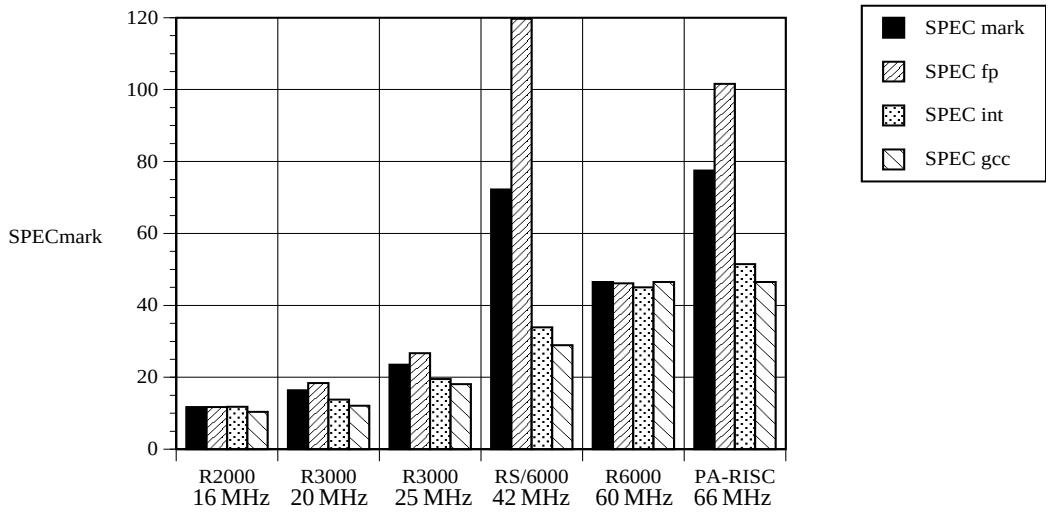

Figure 3: SPECmarks for 6 Processors

By combining monitoring hardware with light software instrumentation, we achieve high precision measurements with minimal system distortion. The only changes to the system were markers inserted into the OS. Markers require about 8 to 12 instructions which take from 24 to 48 cycles to complete. Since few markers are inserted, the kernel's size changed by less that 1%. Likewise, data measurements with and without markers showed little variation in behavior.

4 Experiments and Results

This section presents some of the initial experimental data we have obtained with Monster. We have focused on collecting information about benchmarks that has previously been difficult to analyze using other tools. This includes benchmarks that spend a significant amount of time in the operating system and those that do not seem to benefit from recent trends in computer architecture. A common benchmark program that meets these criteria is a compiler. We will motivate this study by considering recent trends in computer architecture and benchmarking.

4.1 Architectural and Benchmarking Trends

Figure 3 shows the SPECmark ratings for six different machines, each with increasing CPU cycle times. The Y-axis unit is SPECmarks, which is a ratio of execution time of a benchmark versus a reference time measured on a VAX-780. The SPECmark is the geometric mean of all ten SPEC benchmarks combined, while the SPECint and SPECfp numbers are composed from the geometric means of the 4 integer and 6 floating point benchmarks, respectively. We have also plotted separately SPECgcc, the SPEC ratio of the gcc compiler integer benchmark. This study is primarily interested in the gcc benchmark

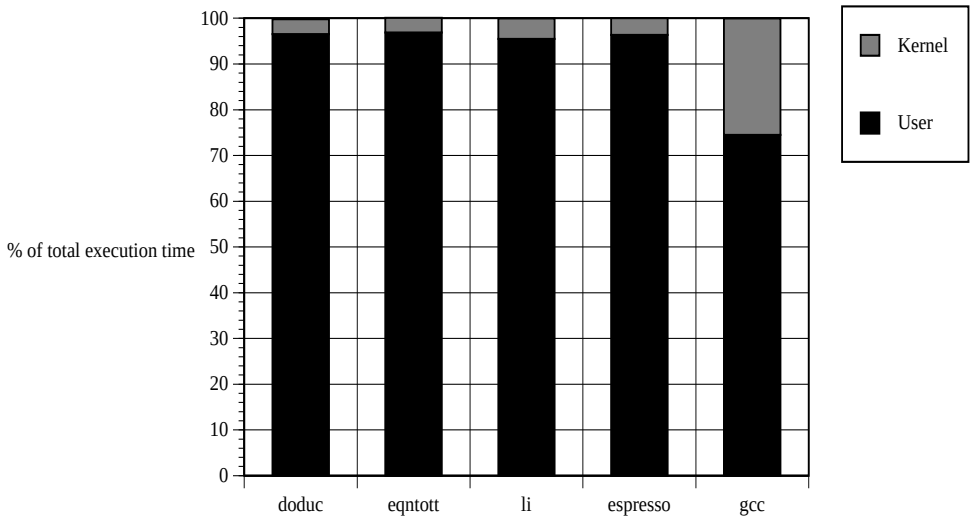

Figure 4: Time spent in user and kernel mode

because it shows the least improvement of all the benchmarks in the suite, and because it is the most difficult to analyze because of the time it spends kernel mode.

The benchmark data in Figure 3 show some clear trends. First, floating point performance tends to be much better than the overall SPECmark. This is particularly true for the RS/6000 and the PA-RISC based systems. Second, performance of the integer SPEC benchmarks consistently lags behind the overall SPECmark. This is most pronounced for the SPECint rating of the RS/6000. Finally, and of most relevance to our study, SPECgcc performance tends to fall below the SPECint average.

In the remainder of this section, we analyze the reasons for poor gcc performance and extend this analysis to other compiler workloads. Our technique permits us to view the entire execution of a benchmark, despite the fact that a good portion of it might execute in kernel mode. Furthermore, Monster enables us to take a hardware view that includes measurement of events such as cache misses, write buffer stalls and floating point unit stalls. This data would be unattainable by using purely software based monitoring techniques.

4.2 An Analysis of Compiler Workloads

All of our experiments were run on a DECstation 3100 using a lightly instrumented version of Ultrix as described in Section 3. Because of variability in some of the measurements, experiments were run multiple times to obtain several samples. Only the average values are reported.

Figure 4 shows the first step of our analysis. Monster was programmed to measure the time a given workload spends in either user or kernel mode. One floating point SPEC benchmark (doduc) and all four of the SPEC integer benchmarks (eqntott, li, espresso and gcc) were selected for this experiment. The results show that most of the SPEC benchmarks spend very little time in the operating system. The one major exception is gcc which spends approximately one quarter of its execution time in kernel mode. For doduc, eqntott, li and espresso, these results justify an analysis restricted to just user mode; the kernel portions of their execution will have little effect on overall performance. An analysis of gcc, on the other hand, could miss some significant effects if it only considered user mode. Referring back to Figure 3, this is precisely what has happened. Performance of newer machines on floating point benchmarks such as doduc and several of the integer benchmarks has steadily increased, while performance on gcc has lagged behind.

Before we proceed, some explanation of how the gcc benchmark works is in order. A typical C compiler consists of three basic phases: a source code pre-processing phase, a compile phase which generates object modules, and a linking phase that combines the object modules into an executable file. The SPEC gcc benchmark only performs part of the middle compile phase on already pre-processed input files. It's output is a collection of Sun assembly language files, so the final linking step is not performed.[1] Since we were interested in a more complete representation of the compilation process, we designed three other compilation benchmarks: cc1, cc2 and cc3. cc1 uses the MIPS cc compiler to preprocesses and compile all of the C source files for the SPEC espresso benchmark and then links the resulting object modules into a final executable. cc2 compiles just one of the espresso object modules and then links all of the espresso object modules together. Finally, cc3 simply links all of the espresso object modules into an executable file.

The same experiment that measured the time spent in kernel and user mode was performed on these three additional compiler workloads (Figure 5). The data clearly show that benchmarks that spend less time compiling and more time linking (cc2, cc3) tend to spend more time in the kernel. The data also show that since the SPEC gcc benchmark consists of only the middle phase of the compiler, it strips away a good portion of kernel execution time that would have to be spent for a real compilation to a final executable file.

The next experiment decomposes kernel execution by determining in which regions of code the kernel spends most of its time. This is the same sort of function that a profiling utility provides when analyzing user applications. These measurements were made by adding markers at the entry and exit points of various regions of code. Monster was then programmed to detect these markers and turn its counters on or off as appropriate.

The regions of execution shown in Figure 6 include the kernel idle loop, `bcopy`, `bzero` and `bcmp`. The kernel idle loop is entered whenever there is no ready process waiting to

1. Only performing the middle phase of a compilation and stopping at assembly language files makes the SPEC gcc benchmark easier to port to other machines.

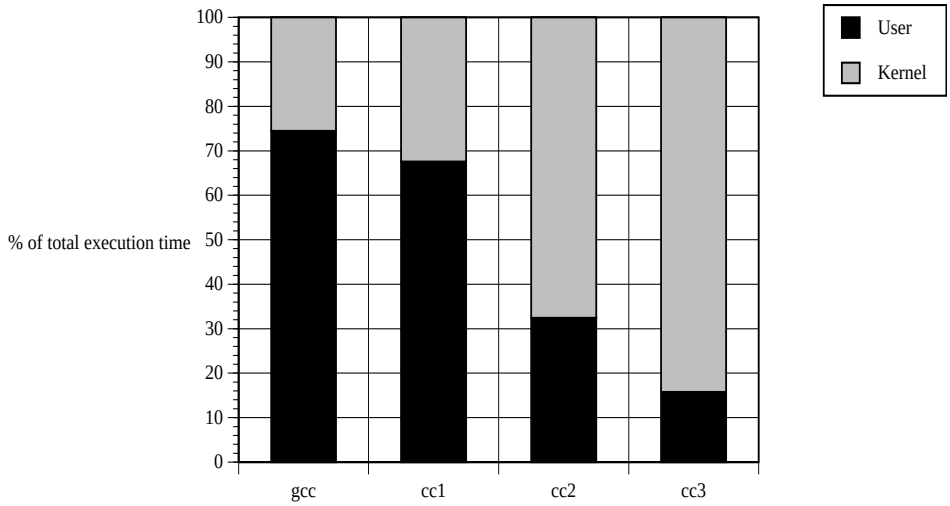

Figure 5: Time spent in kernel for compile workloads

execute on the process run queue. The primary reason for this condition is that the machine is waiting for an I/O transfer to complete. This provides the first clue to explaining the SPEC gcc benchmark's lagging performance; it is more I/O bound than the other SPEC benchmarks. Note that the SPEC gcc benchmark removes a good deal of I/O operations when compared with real compilation benchmarks that generate executable files. Evidence for this is given by the greater proportion of time spent in the idle loop with cc2 and cc3. The other three regions of execution: `bcopy`, `bzero` and `bcmp` are memory block operations that copy, zero fill and compare regions of memory, respectively. Later, we will see that although they comprise relatively little of the kernel's execution time on the 16MHz DECstation, there is reason to believe that future machines will spend significantly more time performing these operations.

It should be noted that the experiments described so far could also be performed using software methods alone. For example, the data in Figure 4 and Figure 5 could be obtained through the use of the UNIX `time` command. Approximations to the data in Figure 6 could be obtained by embedding profiling code in the kernel. But both of these software-only techniques are subject to some error and excessive profiling can result in distorting the kernel's true behavior. Monster's light software instrumentation, combined with external hardware monitoring result in measurements that are precise to a nanosecond and cause little distortion of kernel behavior. The data from the next experiment gives an example of results that cannot be obtained by using software-only techniques.

Figure 7 shows the average cycles per instruction (CPI) of different benchmarks in user and kernel mode. These measurements were taken by counting the number of run cycles and stall cycles in each region and then applying the formula:

$$CPI = \left(\frac{runCycles + stallCycles}{runCycles}\right)$$

Because the idle loop cycles do not represent useful computation, they were subtracted from the run and stall cycles used to compute the kernel CPI [Emer and Clark 84].

The data in Figure 7 expose several points. First, on a DECstation 3100, the CPI of a floating point dominated code (such as doduc) is high. This shows the room for floating point performance improvement over the VAX-780[1] that processors like the RS/6000 and the

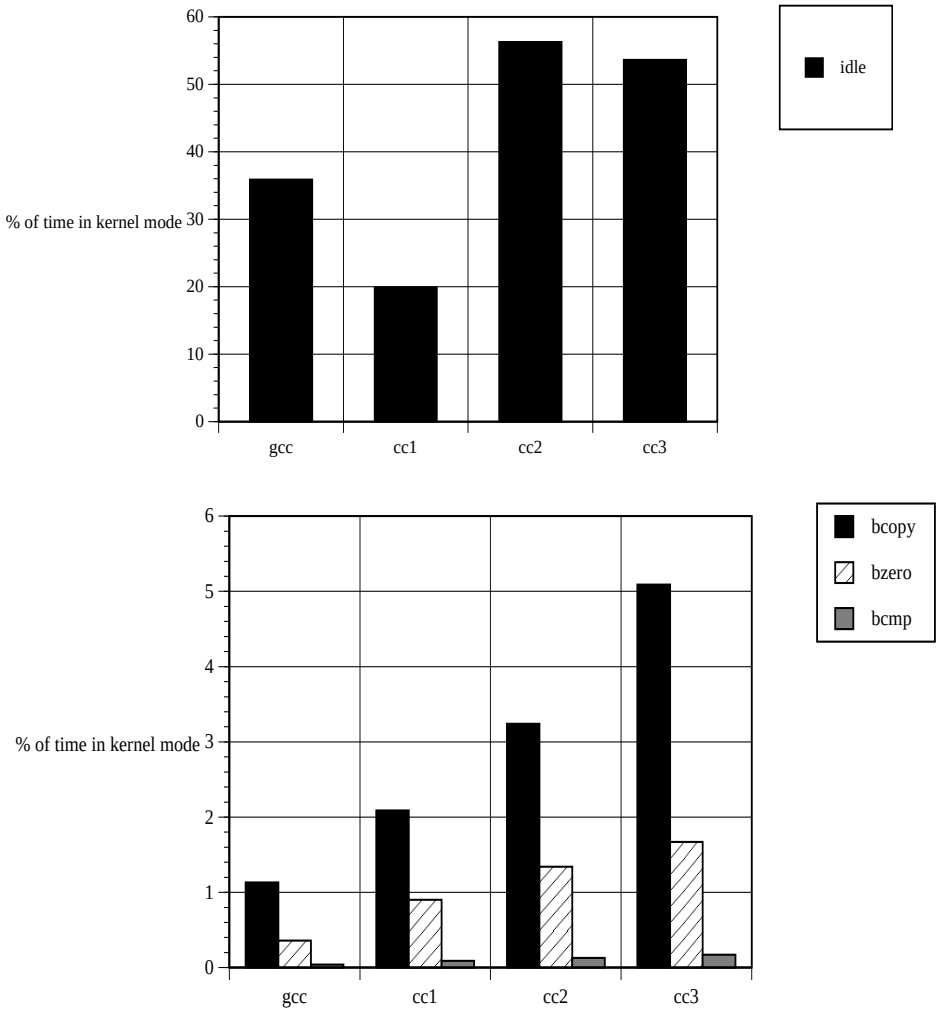

Figure 6: Decomposition of Time Spent in Different Kernel Regions

PA-RISC have exploited. Second, CPI for the integer codes (eqntott, li, espresso and gcc) is low. Even with superscalar instruction issue, processors like the RS/6000 have had difficulty improving integer code performance. A puzzling fact is that with a user CPI of 1.50, gcc seems the most open to performance improvements. Yet, as previously noted, gcc performance is scaling the worst with respect to processor speeds. The final point is that kernel CPI is consistently higher than user CPI for all of the benchmarks. This is of most significance to the compile benchmarks which spend the greatest portion of time in the kernel. Applying Amdahl's law to these results allows us to conclude that as CPUs get faster and faster, they must spend greater proportions of time in kernel mode because of this CPI differential.

The results from the next experiment show one of the reasons why kernel CPI tends to be higher than user CPI. Figure 8 shows CPI in the kernel `bcopy`, `bzero` and `bcmp` regions of execution while running the four compile benchmarks. The measurements expose a memory bandwidth problem on a machine that operates at only 16 MHz. Future systems with cycle times 10 times this rate can expect to spend significantly more time in these regions of execution unless architects pay more attention to the interface between main memory and the cache.

The next experiment shows precisely which hardware resources are the bottleneck for different types of codes (Figure 9). We classified stalls into four basic categories: write stalls, read stalls and floating point unit stalls. Read stalls are caused by instruction and data cache misses which result in a 5 cycle penalty to access main memory. Write stalls occur

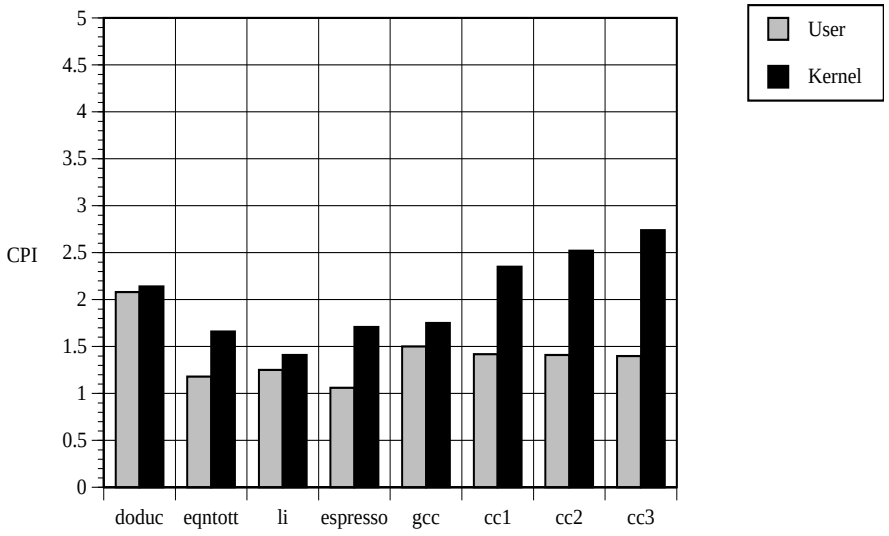

Figure 7: CPI in user and kernel mode

1. The VAX-780 is the SPECmark reference machine.

when the 4-entry write buffer fills to capacity. The DECstation 3100 uses a write-through policy, so all stores must be placed into the write buffer before being retired to main memory at a rate of one word per 5 cycles. Finally, floating point stalls are due to the floating point coprocessor.

Figure 9 shows that the kernel stalls primarily on memory requests. Thus, improving floating point performance will do nothing to improve overall kernel performance. The data

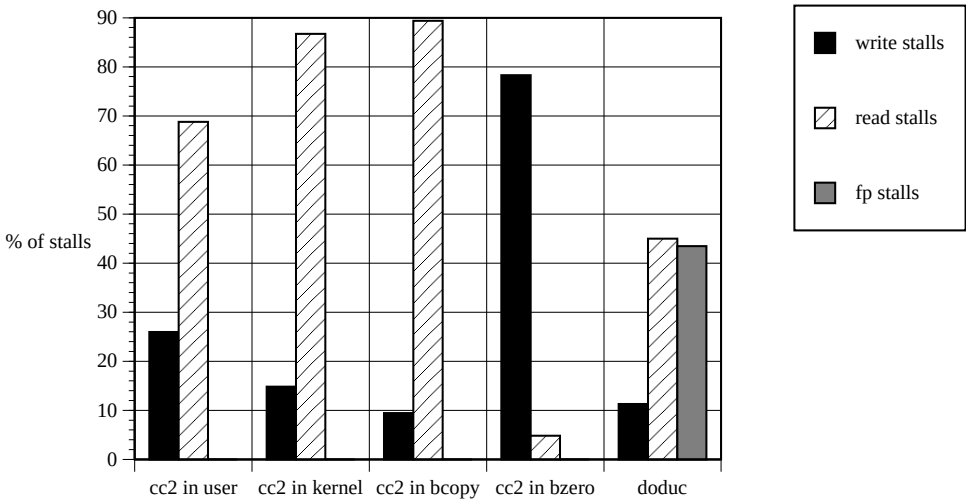

Figure 9: Stall Breakdown (percent)

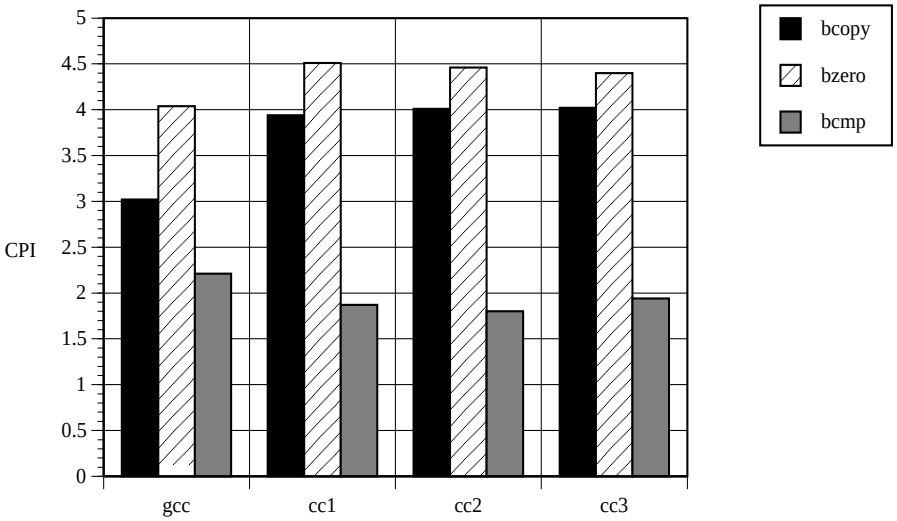

Figure 8: CPI in different kernel regions

also provide more information regarding performance bottlenecks in `bcopy` and `bzero`. Figure 9 shows that in `bcopy`, the limiting factor is memory reads which account for about 90% of `bcopy` stalls. In `bzero`, on the other hand, the bottleneck is with memory writes which account for nearly 80% of stalls in that region. To clear out an entire region of memory, `bzero` quickly fills the write buffer which limits performance to the rate at which stores can be retired to main memory. This explains `bzero`'s 4.5 CPI value. Although `bzero` and `bcopy` only account for about 1.5% and 5.0% (respectively) of total execution time on the DECstation 3100, a machine with a processor that is 10 times as fast could spend a significantly greater portion of time performing this function if its memory bandwidth does not increase proportionately. To avoid this, architects need to consider ways to provide higher main memory bandwidth or perhaps move to write-back caching policies, despite the memory coherency complications that they cause.

For the regions of execution that contain large percentages of read stalls, we performed another experiment to further decompose this activity (Figure 10). We classified read stalls

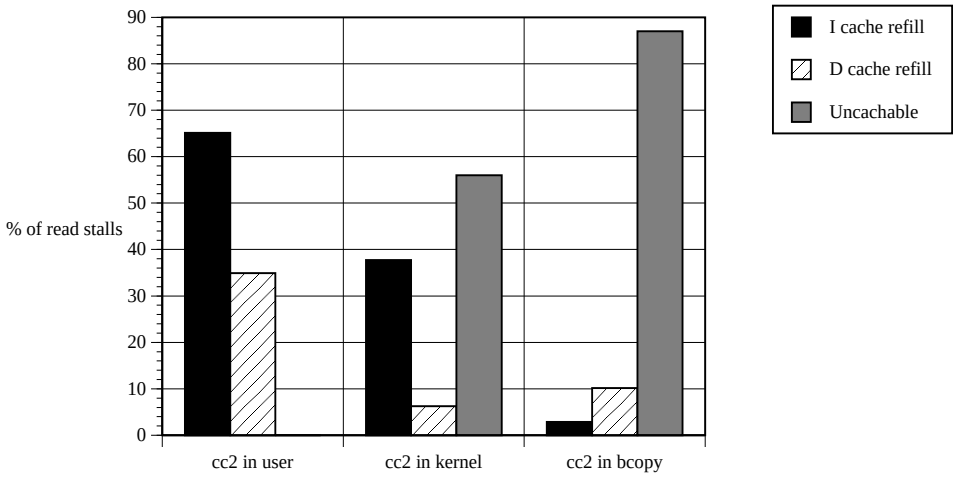

Figure 10: Read Stall Breakdown (in percent)

into three categories: those that result in an instruction cache line refill, those that refill a data cache line and those that are associated with references to non-cacheable memory.[1] The most notable data in Figure 10 are the high percentages of uncacheable references when running in the kernel in general and in `bcopy` in particular. Over 80% of the reads performed by `bcopy` are not eligible for caching. This once again underscores the impor-

1. All references from user mode are cacheable, but the kernel specifies regions of memory (I/O buffers in particular) that are not cacheable.

Type of Instruction	cc2 user	cc2 kernel
nops	19.46	26.06
jumps	3.50	6.20
branches	11.37	10.28
loads	24.10	17.78
stores	11.18	14.16
alu	30.02	23.60
integer multiply and divide	0.06	0.25
floating point	0.00	0.00
system	0.00	1.86
marker	0.00	0.49

Table 1 : Instruction Mix for cc2 (in percent)

tance of improving main memory bandwidth, or perhaps considering some form of snoopy DMA which preserves cache consistency when performing I/O.

Our last experiment was to study the relative frequency of instructions in user and kernel mode to determine if there are any differences. Table 1 shows an instruction mix for cc2 when executing in user and kernel mode. The measurements were made by programming Monster to detect and count the frequency of different classes of instructions when executing in either user or kernel mode. The same measurements where taken while executing in the idle loop (not shown here) so that the idle loop instructions and cycles could be subtracted from the other kernel instructions. In other words, the kernel mix shows the relative frequency of instructions executing in kernel mode, but not in the idle loop.

The numbers reveal several differences between the dynamic frequency of instructions in kernel mode and user mode. First, the kernel executes 6% more `nop`s than user code does. Over one fourth of all kernel instructions are `nop`s. Second, the kernel does not seem to execute significantly more branch or jump instructions, at least relative to a compile benchmark. Third, in user mode the ratio of loads to stores is close to 2:1, while in kernel mode it is closer to 1:1. This reflects the kernel's role as a "mover of data" in contrast with user code's role of operating on that data. This observation is also supported by the relative infrequency of ALU, floating point unit and integer multiply/divide operations in the kernel. Finally, notice that the frequency of marker instructions added for the purpose of

triggering supports our claim that the impact of our instrumentation to the kernel is very slight.

5 Summary

The previous section has shown the sort of system data that Monster is capable of obtaining. The analysis of compiler workloads was performed to validate our approach and has exposed some of the reasons for poor compiler performance relative to other applications. These include:

- Kernel CPI is almost a full point higher than user CPI. An analysis of stall types showed that high kernel CPI is due mostly to a very high CPI in `bcopy` and `bzero` caused by inadequate main memory bandwidth and a write-through caching policy.
- When in kernel mode, a significant portion of time is spent spinning in the idle loop, waiting for disk I/O to complete.
- Even when disk blocks are found in the file block cache, they must be copied from uncachable memory. This results in a surprisingly high number of uncachable memory references which place a bound on the maximum achievable hit rate for the CPU caches.
- Instruction mixes show some interesting differences between the kernel and user modes of execution. In particular, the ratio of loads to stores shows that the kernel functions primarily as a “mover of data”. This is an indication that the perhaps the best way for hardware to support the operating system is to make it easier to quickly move data from place to place.

With our future work, we hope to examine more complex interactions between architectures and operating systems. Part II of this report presents in greater detail our immediate plans.

Part II - Proposal for Future Work

1 Introduction

Monster was developed because of our interest in architectural support for operating systems. We would like to study the needs of applications which rely heavily on operating system services and then determine what hardware features best support these needs. Our work is motivated primarily by [Anderson et al. 91] and [Ousterhout 90] who argue that operating systems and computer architectures are evolving in somewhat incompatible directions.

To get started, we will restrict the scope of our studies to improving application performance with respect to a machine's instruction and data CPU caches. In particular, we will focus on complex applications. When we say *complex application*, we mean a program which is implemented using multiple address spaces, multiple threads of control and tends to spend a large fraction of time executing operating system code. We contrast this with other cache studies that consider "simple" applications which use only one address space, one thread of control and spend little time invoking operating system services.[1]

We will focus on reducing conflict[2] misses in both the instruction and data caches by using two basic techniques:

(a) The use of page allocation policies to carefully map pages into CPU caches. If a cache is physically addressed and is larger than the virtual memory page size, then the selection of physical page frames determines where data will be physically placed in the cache. A careful page allocation policy attempts to evenly distribute pages in the cache to minimize potential cache conflicts.

(b) The use of static analysis and dynamic execution profiles during code generation to carefully place instructions and data in CPU caches. A compiler that knows which instructions and data are frequently accessed could use this information to spread out references to the cache (in space and time) and thus reduce conflict misses.

Technique (a) is most naturally implemented by an operating system, while technique (b) is best realized by a compiler. Throughout this paper, we will refer to technique (a) as *care-*

1. A typical example of application of this sort can be found in the SPEC 1.0 benchmark suite studied in Part I. Each of these applications runs in a single UNIX process and usually makes very few system calls. [SPEC 91]
2. Conflict misses occur when an item is evicted from the cache by a reference to a different item, but then must be re-fetched later on. This is in contrast with compulsory (cold-start) and capacity misses.

ful mapping, while technique (b) will be referred to as *careful placement*. We also plan to study ways to combine careful mapping and careful placement:

(c) Information stored with the pages of an executable image could be used by the page allocation algorithm to avoid overlapping *incompatible*[1] pages. When forced to map pages that will overlap in the cache, the operating system could use information provided by the compiler to select pages that minimize cache conflicts. Information that would guide page placement might include marking portions of the page which are "hot spots", i.e. those portions of the page which are likely to be frequently accessed.

The rest of this proposal motivates this work, summarizes previous work in the area and describes the above techniques in greater detail. We then present a plan for implementing these techniques and finally conclude with the possible impact of the work.

2 Context

Caching data and instructions in fast memories close to the CPU has long been recognized as an effective technique for improving performance. Recently, the importance of this technique has grown due to the dramatic decrease in CPU cycle times. In a next generation computer system, it is possible that a memory access which misses the cache could result in a penalty of hundreds of CPU cycles to fetch the data from main memory [Olukotun et al. 91] [Hennessy & Jouppi 91]. This trend underscores the importance of finding new techniques to more effectively use CPU caches.

2.1 Previous Work

The policies that dictate where data is placed in a CPU cache are usually implemented in hardware. Because of the speed at which decisions must be made, the policies tend to be very simple. For example, it has been suggested that a direct-mapped scheme is often the most effective placement policy for a CPU cache [Hill & Smith 89]. But many researchers have observed that even higher performance can be obtained by combining the various software based techniques (such as those discussed in the introduction) with a simple hardware implemented cache placement policy.

The conflict misses caused by multiple processes competing for room in a cache has been studied by [Stone & Thiebaut 86] and [Mogul & Borg 91] using mathematical analysis and trace-driven simulation. In [Kessler & Hill 90], trace-driven cache simulations are used to investigate different page mapping policies designed to minimize conflict misses from overlapping pages in the CPU cache. This work corresponds to technique (a) (careful mapping) as described in the introduction. Although the policies described by Kessler attempt to minimize conflicts within a single address space, they could be extended to minimize

1. Pages are more *compatible* with each other if their contents are not likely to interfere with each other (in space or time) during the execution of the application.

conflicts between the multiple address spaces of several processes. In [Taylor et al. 90], a heuristic called page-coloring is used to assign page frames so that physical pages tend to be distributed in the same way that virtual pages are. Page coloring is actually one of the heuristics that Kessler uses.

In [Hwu & Chang 89], dynamic execution profiles are used by an optimizing compiler to place instructions in a way that minimizes conflicts in the cache. Experiments based on trace-driven simulation show that miss rates for carefully placed code in a direct mapped instruction cache (I-cache) are consistently lower than unoptimized code in larger I-caches and I-caches with greater degrees of associativity. In [Lam et al. 91], static code analysis and blocking algorithms are used to re-order accesses to large data structures (e.g. large arrays) so that they are operated on in chunks or blocks. This reduces conflict misses in the data cache (D-cache) due to repeated reloading of array items. Experimental results (again, based largely on simulation) show speed-ups of between 3 and 4.3 on matrix multiply code. This compiler based work corresponds to technique (b) (careful placement) from the introduction. Hwu's work focuses on I-cache improvements, while Lam's work concentrates on D-cache improvements.

2.2 Our Contributions

We would like to extend the previous work described in Section 2.1 in several ways.

First, most of the previous work uses purely simulation models or mathematical analysis. We would like to validate the previous work with experimental data from an actual machine. Monster enables us to perform these experiments because we can directly measure I-cache and D-cache misses (among other hardware events) on a DECstation 3100.

Second, many of these previous studies have been restricted to applications which spend very little time executing operating system code (or they factor OS code out) and consist of only a single UNIX-style process (one address space and one thread of control). Some studies have considered the effects of multiple processes, but they neglect the OS code that schedules and manages the processes. There are many interesting complex applications that consist of multiple address spaces, multiple threads of control and spend significant portions of time executing operating system code. We intend to examine these complex applications in our studies.

Third, to our knowledge, no one has studied technique (c) which combines careful mapping and placement. We believe that in order for careful mapping and placement to work well in actual systems, the operating system and compiler must cooperate. For example, we anticipate potential problems with Lam's blocking techniques if the operating system maps different virtual pages onto the same physical cache page, thus thwarting the compiler's careful data placement attempts. To avoid this, the operating system must be made aware of the compiler's efforts. We also believe that only if careful mapping and placement are integrated will they be applicable to the complex applications we wish to study.

Finally, we would like to pay special attention to the operating system kernel. For example, we would like to study the effects of dedicating portions of the D and I-caches solely to the kernel. This is a variation of careful mapping. Also, because Monster can monitor execution in kernel mode, we can use technique (b) to profile and then carefully place and optimize the execution of frequently referenced kernel code and data. Because the kernel plays a role in the execution of all applications, it makes sense to apply extra effort to optimizing it. A faster kernel will automatically make many applications run faster.

3 More Details

3.1 Technique (a): Careful Page Mapping

A basic design choice for operating systems that implement virtual memory is the page replacement policy. Page replacement involves selecting a page to evict when main memory starts to get full. Usually the new page is mapped to a physical page frame according to an approximation of a LRU or MRU policy and without regard to where the page will reside in the CPU cache. The goal of careful mapping is to slightly modify the page replacement policy so that the CPU cache *is* taken into consideration.

In a direct-mapped cache, multiple main memory page frames map to the same physical cache page frame. So, cache page frames can be viewed as "bins" for holding multiple page frames. By distributing pages among the cache bins as evenly as possible, the frequency of cache conflicts can be minimized. For example, Figure 11 shows two possible mappings of virtual pages to physical page frames. In the first mapping (a), some cache bins are not mapped at all (thus rendering entire portions of the cache useless), while other cache bins are over-utilized (which is likely to cause more cache conflicts). The second mapping (b) evenly distributes pages among the cache bins and is clearly more desirable. The ultimate goal of careful page mapping is to arrive at distributions more like the second type.

3.2 Technique (b): Careful Instruction and Data Placement

Careful memory use is an important class of compiler optimization techniques. A compiler that understands the memory system can often use this knowledge to advantage. For our studies, we are interested in optimizations that involve an understanding of either a machine's I or D-cache.

For example, a compiler that knows about the I-cache can use the following instruction placement algorithm[1]:

(1) An execution profile of the program is obtained.

(2) Using the execution profile, a weighted call graph is generated. A *call graph* is a directed graph where every node is a function (procedure) and

1. Borrowed from [Hwu & Chang 89].

every arc is a function call. A *weighted call graph* is a call graph in which all the nodes are marked with their execution frequencies.

(3) The weighted call graph is used to place functions with overlapping lifetimes into memory locations which do not contend with each other in the cache. This has the effect of reducing cache mapping conflicts.

Figure 12 shows a weighted call graph and two possible instruction placements. The call graph shows that function `f()` repeatedly calls function `g()`, perhaps from within a loop. Thus, the instructions that make up `f()` and `g()` will repeatedly be fetched close together in time. The figure also shows two instruction placements. In the first placement (a), the compiler has positioned the functions in the upper portion of separate pages so that they could overlap in the cache. Usually, the operating system will map the pages into different cache bins (c), but if the pages are mapped into the same cache bin, as in situation (d), then the two functions will conflict in the cache and performance will drop dramatically. Although this situation is unlikely, when it does occur, the penalty can be severe; nearly every instruction fetch may have to go to main memory. The compiler can avoid this problem by using the simple heuristic of placing the two functions on the same page, as shown

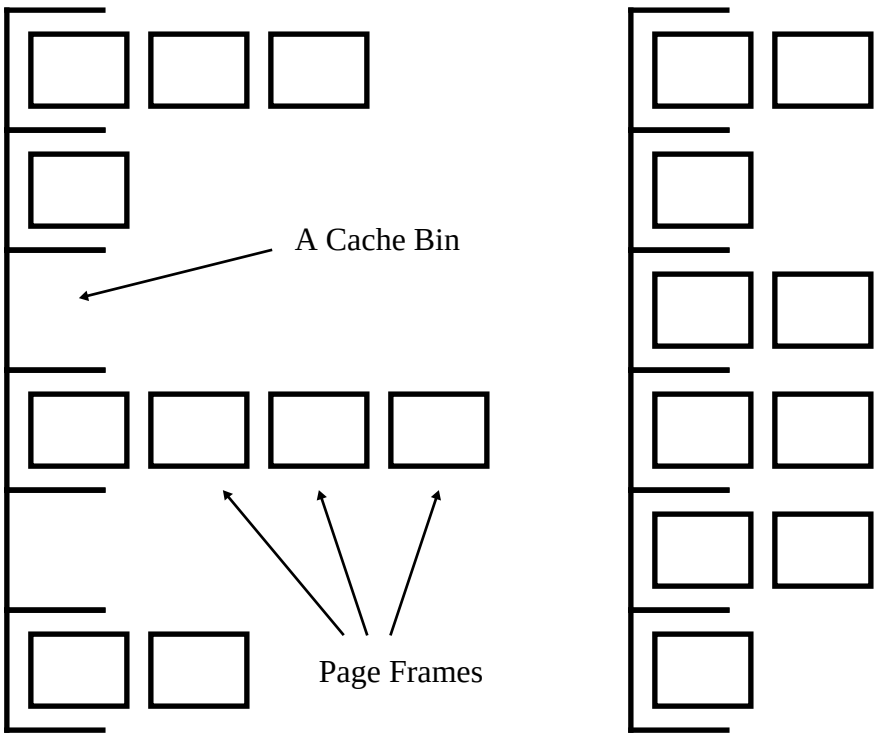

Figure 11: Two Possible Mappings of Pages into a Physical Cache (Borrowed from [Kessler & Hill 90])

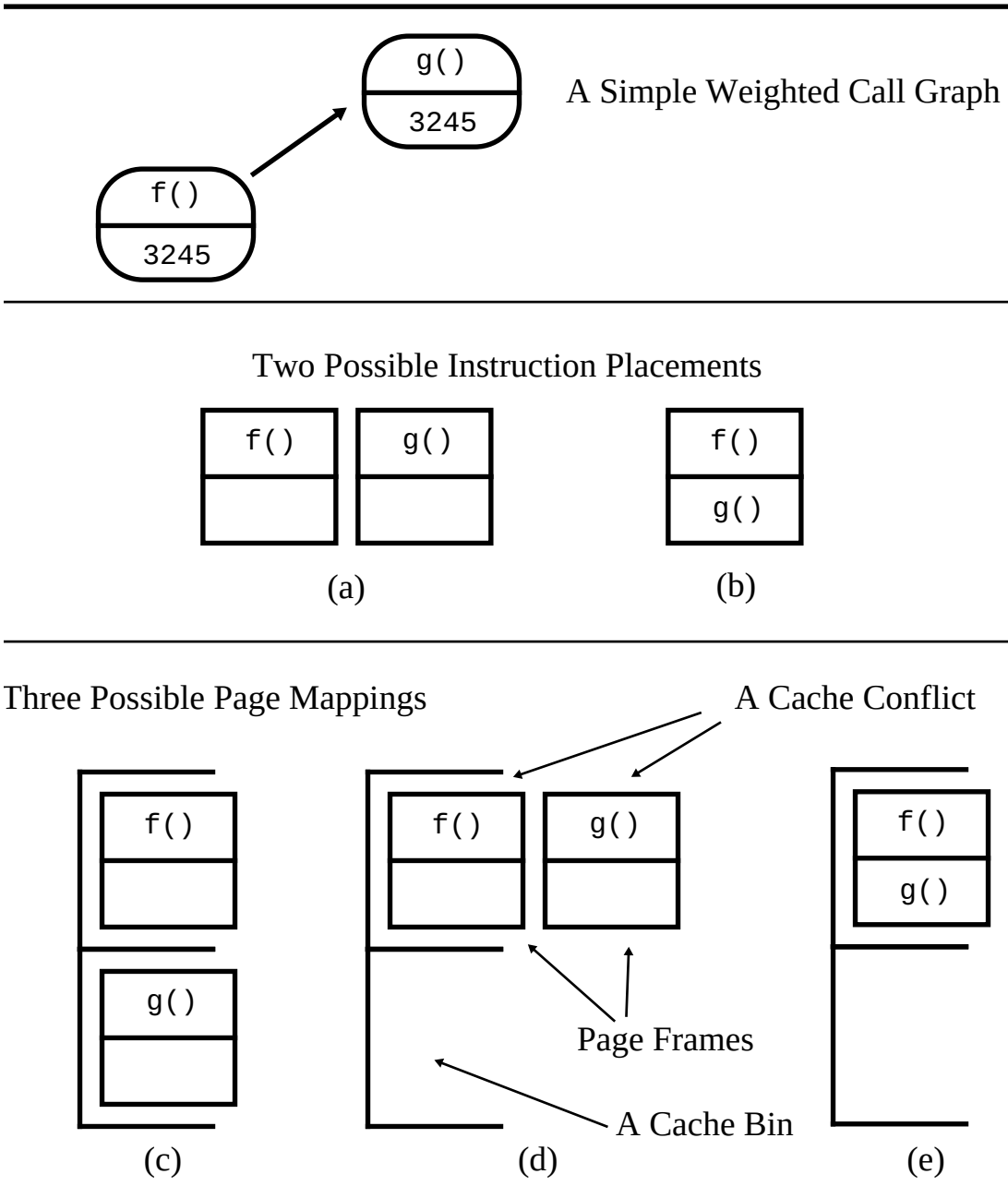

Figure 12: Examples of Instruction Placement Options

in placement (b). This ensures that they won't conflict in the cache as shown in placement (e).

As a second example, consider a compiler which knows about the D-cache. Suppose that static analysis reveals that an array larger than the D-cache is being accessed in some reg-

ular striding pattern (say for a matrix multiply). Without optimizations, different parts of the large array would continually be re-fetched into the cache. An optimization known as "blocking" or "tiling" can avoid this situation by doing the following[1]:

(1) Reorganize (block) the data so that only a small section of a large data structure is loaded into the D-cache at a time.

(2) Operate on the blocked, cached portion of the data structure.

(3) Block another portion of the data structure into the cache and repeat (2).

Note that blocking can only be applied to very regular and predictable code such as that typically found in scientific applications. Also note that if the operating system maps pages into the cache without regard for the compiler's carefully data placement efforts, much of the performance gain could be lost.

3.3 Technique (c): Integrating Careful Page Mapping and Placement

The description of techniques (a) and (b) was derived from what we know about the work of other researchers. To our knowledge, these techniques have only been studied in isolation. There have been no studies of interactions between these optimizations, nor of the possibility of integrating them together. There are at least three reasons why we think such a study would be interesting:

- **By making the operating system and the compiler aware of each other's optimization efforts, they are less likely to interfere with each other.** For example, if the OS knows that the compiler has based an optimization on the assumption that two virtual pages will not overlap physically in the cache, then the operating system can try to make the assumption true.
- **By sharing information, the operating system and compiler can simplify each other's optimization efforts.** For example, if the OS knows that two virtual pages are unlikely to interfere with each other in the cache (i.e., they are compatible), it has some additional freedom when mapping pages (namely, it can overlap the compatible pages in the same cache bin without concern for performance). Or, if the compiler can mark two pages as incompatible, then it can be reasonably certain that instructions from those pages won't conflict in the cache and relax its attempts to group related instructions on the same page.
- **Integrating the operating system and compiler's optimizations enables more complex applications to benefit from careful placement optimizations**. Although many older applications are constrained to a single UNIX-style process, some more recent applications utilize multiple address spaces and threads of control. However, because the compiler's careful

1. As suggested by [Lam et al. 91] and others.

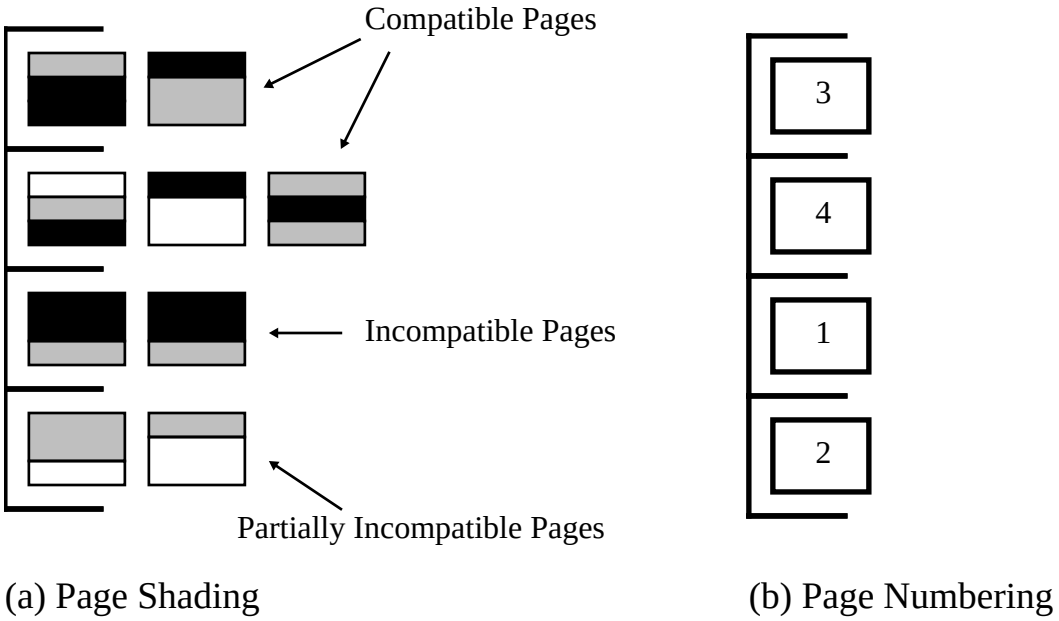

Figure 13: Some Page Marking Schemes

placement optimizations operate within a single address space, they are incapable of avoiding conflicts between address spaces. This may become an important limitation if the trend to decompose programs into multiple address spaces continues to be used as a program structuring technique. To overcome this problem, the compiler could mark specific pages according to some "universal compatibility policy". Then, the operating system can dynamically map the pages of many different addresses spaces into the cache so that conflicts are minimized.

The above points argue for cooperation between operating system and compiler based cache optimizations. But how can the OS and compiler share information about their optimizations? We propose two mechanisms by which the compiler could communicate information to the OS: *page numbering* and *page shading*.

Figure 13 shows how compiler hints might be used by the OS to improve page mapping. With page shading, different fragments[1] of a page would be shaded[2] by the compiler to indicate compatibility. Two page fragments are considered compatible if they are shaded differently. The first two cache bins in Figure 13(a) show page mappings that are com-

1. The pages are divided into thirds for this example.
2. This example uses three shades: black, grey and white.

pletely compatible. The third cache bin, on the other hand, shows a mapping in which pages are completely incompatible. Varying degrees of page compatibility are also possible, as shown by the mapping in the fourth cache bin. Page shading gives the compiler a way to tell the operating system which pages should *not* be overlapped in the cache or which pages could freely be overlapped without affecting performance.

With page numbering, sets of pages are ordered in a sequence ranging from 1 to N, where N is the number of cache bins. The operating system would then attempt to order these pages in the same sequence when it maps them to cache bins. Note that the sequence need not begin in the first cache bin. It can begin in a middle bin and then "wrap around" the end of the cache. Page numbering would be useful for a compiler that uses data blocking techniques and relies on pages being spread out in the cache.

4 Approach

This section explains how we plan to proceed with this work. Our basic strategy is to borrow hardware and software developed by others as much as possible. We are interested in studying different optimization techniques and their interactions, not in completely reimplementing an operating system and compiler. Section 4.1 lists the hardware and software that we have and Section 4.2 describes how we hope to pull everything together.

4.1 Hardware, Software and Other Tools

We have a DECstation 3100 (nicknamed Frankenstein) with the following components:

- A direct mapped, physically addressed 64K D-cache.
- A direct mapped, physically addressed 64K I-cache.
- A TLB that supports 4K pages.
- 3 disk drives.

These parameters show that Frankenstein is well suited for the studies. First, each cache can be viewed as having 16 bins for holding physical pages. There are two caches (instruction and data) for a total of 32 cache bins, and because the caches are physically addressed, they are suitable for careful page mapping algorithms. Second, since the caches are direct-mapped, they are ideal for careful placement techniques. Third, the three different disk drives enable us to easily rebuild operating system kernels and to boot the machine under different operating system versions (Ultrix, OSF/1 and Mach 3.0). Finally, the motherboard of the machine has been modified to make it accessible to Monster.

We also have access to the following operating systems, compilers and optimization tools:

- Source code for an old version of Ultrix.
- Source code for OSF/1 (Mach 2.5).

- Source for the Mach 3.0 kernel from CMU.
- The `pixie` MIPS code profiler.
- An optimizing MIPS compiler which accepts `pixie` profile data.
- Source code for the GNU `gcc 2.0` compiler.

Both OSF/1 and Ultrix run on Frankenstein. We have studied portions of the Ultrix and OSF/1 sources (`locore.s` in particular) well enough to be able to rebuild a kernel with markers installed. We are currently porting Mach 3.0 and a user-level UNIX server on Frankenstein. Having access to the source code for these different operating systems enables us to implement the careful page placement algorithms.

The `pixie` profiling tool uses program annotation techniques to construct dynamic execution profiles. The MIPS optimizing compiler can be invoked with a special `-cord` option which uses this profile data to carefully place instructions in the program's virtual address space to improve cache performance. This tool provides us with a full implementation of careful instruction placement.

The GNU `gcc 2.1` compiler generates code for MIPS based machines (among others) and has many procedural hooks for influencing the code generation phase. Because the source code is freely available and is reasonably well documented, we hope that it might serve as a starting point for implementing careful data placement (blocking) algorithms.

4.2 Pulling Things Together

As you can see, we have a hodgepodge of hardware and software at our disposal. Some of it can easily be adapted to perform our experiments, but some will require moderate to extensive modification to meet our needs. This section specifies the experiments we plan to perform and the necessary modifications to our tools.

4.2.1 Experiments

Here are the different classes of experiments we would like to perform:

- **Experiment Class 1: Studies of careful page mapping in isolation**. We will implement the different mapping policies proposed by [Kessler & Hill 90] and then use Monster to measure changes in cache hit rates.
- **Experiment Class 2: Studies of careful page placement in isolation**. We will implement the placement algorithms proposed by [Hwu & Chang 89] and [Lam et al. 91] and then use Monster to measure changes in cache hit rates.
- **Experiment Class 3: Studies on integrating careful page mapping and placement**. We will implement our proposed page shading and page numbering mechanisms so that the compiler and operating system cooperate to

improve cache performance. Different policies involving the granularity and levels of shading will be evaluated by measurements made with Monster.

- **Experiment Class 4: Studies of kernel specific optimizations**. We will use the page mapping mechanisms developed for experiment Class 1 to investigate policies such as reserving portions of the cache for the kernel. Also, we will use Monster to profile execution of the kernel and then use the careful page placement mechanisms developed for Class 2 experiments to optimize kernel execution in the I-cache.

4.2.2 Implementing the Experiments

To perform the experiments of Class 1, we need to influence the page replacement algorithms on the DECstation. There are two ways we could tackle this:

- For Ultrix, we can modify the page replacement algorithms in the kernel. Specifically, we have looked at some of the code and it appears that the modifications could be made by changing the `memall()` and `vmemall()` routines.
- For OSF/1 (Mach 2.5) and Mach 3.0, we can modify a user-level pager process. We hope to draw on work by [Sechrest & Park 91] and [McNamee & Armstrong 90] in which the Mach external pager interface is extended to allow the page replacement policy to be implemented in a user-level process.

To perform the experiments in Class 2, we need to influence the way a compiler places instructions and data. For instructions, this will be easy. We have a comprehensive set of compiler optimization tools [MIPS 88]. In particular, we can:

- Generate dynamic execution profiles using `prof` and `pixie`.
- Translate the `pixie` and `prof` outputs into the `cord` format by using the `ftoc` tool. The `cord` format is just an encoding of the information needed to construct a weighted call graph. It is used by the MIPS C compiler to carefully place instructions in the I-cache.

Carefully placing data will be more difficult. We don't have any tools that already do this, so we must resort to two basic options:

- Study the `gcc 2.1` compiler. A cursory examination of the documentation seems to indicate that there are many procedural hooks into the code generation phase that might enable us to work in the blocking algorithms.
- Another alternative is to work on source-to-source restructuring tools. For our purposes, it may be sufficient for us to block the code by hand at the source level.

To perform the experiments of Class 3, we need to solve the basic problem of getting page shading or page numbering information from the compiler to the operating system. This could be implemented by modifying the object file format and the loader.

Finally, to implement the experiments of Class 4, we first need to collect some dynamic execution profiles for the kernel. We can't use `pixie` and `prof` for this because they only work on user-level applications. However, we can use Monster to obtain this data. We have studied the `cord` format required by the MIPS C compiler for its instruction placement optimizations. It is a very simple format that should make it easy for us to encode the profile data we obtain with Monster. Reserving portions of the cache for the kernel should also be straightforward once we have the page mapping controls in place.

5 Summary

The second part of this report has presented a collection of optimizations which reduce conflict misses in instruction and data CPU caches. Most of these optimizations have only been evaluated with simulation techniques. We intend to use Monster to obtain realistic data on actual implementations of these optimizations to determine whether and under which circumstances they make sense.

6 References

[Agarwal 89] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Kluwer Academic Publishers, Boston, 1989.

[Anderson et al. 91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad and Edward D. Lazowska. The interaction of architecture and operating system design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108-120, 1991.

[DEC 92] Digital Corporation. Digital 21064-AA Product Brief. Preliminary Release, 1992.

[Emer & Clark 84] Joel S. Emer and Douglas W. Clark. A characterization of processor performance in the VAX-11/780. In *11th Annual Symposium on Computer Architecture*, pages 301-309, 1984.

[Fitzgerald & Rashid 86] R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer systems*, 4(2):147-177, May 1986.

[Groves and Oehler 91] Randy D. Groves and Richard Oehler. RISC System/6000 processor architecture. In *IBM RISC System/6000 Technology*, pages 16-23, 1991.

[Golub et al. 90] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87-95, 1990.

[Hill & Smith 89] M. Hill and A. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612-1630, 1989.

[Hwu & Chang 89] W. Hwu and P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual Symposium on Computer Architecture*, pages 242-251, 1989.

[Hennessy & Jouppi 91] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: an evolving interaction. *Computer*, 24(9); 18-29, September, 1991.

[Kessler & Hill 90] R. Kessler and M. Hill. Miss reduction in large, real-indexed caches. *University of Wisconsin Tech Report*, 1990.

[Lam et al. 91] Monica S. Lam, Edward E. Rothberg and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63-74, 1991.

[Larus 90] James R. Larus. *Abstract Execution: A Technique for Efficiently Tracing Programs*. University of Wisconsin-Madison, Madison, WI, 1990.

[Li & Hudak 89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321-359, November 1989.

[McNamee & Armstrong 90] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17-29, Burlington, Vermont (USA), October 1990. USENIX Association.

[Mogul & Borg 91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75-84, 1991.

[MIPS 88] MIPS Computer Systems, Inc. *RISCompiler Languages Programmer's Guide*, 1988.

[Nagle & Uhlig 92] D. Nagle, R. Uhlig and T. Mudge. *Monster: A tool for analyzing the interaction between operating systems and computer architectures.* University of Michigan Tech Report, 1992.

[Nelson 89] Harry Nelson. Experiences with performance monitors. In *Instrumentation for Future Parallel Computing Systems*, pages 201-208, 1989.

[Olukotun et al. 91] O. A. Olukotun, T. N. Mudge and R. B. Brown. Implementing a cache for a high-performance GaAs microprocessor. In *The 18th Annual International Symposium on Computer Architecture*, pages 138-147, 1991.

[Ousterhout 90] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247-256, 1990.

[Pierce 92] James Pierce. IDTrace for the X86 Architecture. Advanced Computer Architecture Seminar. The University of Michigan, 1992.

[Schroeder & Burrows 90] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1-17, February, 1990.

[Sechrest & Park 91] S. Sechrest and Y. Park. User-level physical memory management for Mach. *Extended Abstract to a USENIX Mach Workshop*. 1991.

[SPEC 91] SPEC Newsletter, 3(3-4), 1991.

[Stone & Thiebaut 86] H. Stone and D. Thiebaut. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305-329, 1987.

[Taylor et al. 90] G. Taylor, P. Davies and M. Farmwald. The TLB slice -- a low-cost high-speed address translation mechanism. In *The 17th Annual International Symposium on Computer Architecture*, pages 355-363, 1990.