Operating System Benchmarking in the Wake of *Lmbench*: A Case Study of the Performance of NetBSD on the Intel x86 Architecture

Aaron B. Brown
Margo I. Seltzer
Harvard University
{abrown,margo}@eecs.harvard.edu

Abstract

The *lmbench* suite of operating system microbenchmarks provides a set of portable programs for use in cross-platform comparisons. We have augmented the *lmbench* suite to increase its flexibility and precision, and to improve its methodological and statistical operation. This enables the detailed study of interactions between the operating system and the hardware architecture. We describe modifications to *lmbench*, and then use our new benchmark suite, *hbench:OS*, to examine how the performance of operating system primitives under NetBSD has scaled with the processor evolution of the Intel x86 architecture. Our analysis shows that off-chip memory system design continues to influence operating system performance in a significant way and that key design decisions (such as suboptimal choices of DRAM and cache technology, and memory-bus and cache coherency protocols) can essentially nullify the performance benefits of the aggressive execution core and sophisticated on-chip memory system of a modern processor such as the Intel Pentium Pro.

1 Introduction

As modern applications become increasingly dependent on multimedia, graphics, and data movement, they are spending an increasing fraction of their execution time in the operating system kernel. A typical web server, undoubtedly today's hottest server application, can spend over 85% of its cycles running operating system code [4]. Other multimedia, commercial, and GUI workloads spend between 20% and 90% of their instructions in the kernel as well [4][6][10]. Amdahl's law tells us that if we want these modern applications to run quickly, the operating systems must run quickly as well. However, most of the performance analysis that has driven processor and system design has been directed towards application-level performance as quantified by the SPEC benchmarks, which have been shown to execute fewer than 9% of their instructions within the kernel [6]. Therefore, if we want to know which hardware to tune to make the OS perform better, thereby improving the performance of OS-dependent applications, it is critical that we understand the hardware and architectural basis of OS performance.

There are three standard approaches to quantifying and understanding OS performance: macrobenchmarking (i.e., running actual applications), profiling, and microbenchmarking. Macrobenchmarks, although useful for measuring end-to-end performance on a specific workload, involve too many variables to form a foundation for understanding the hardware basis of OS performance. Kernel profiling can be useful for determining the performance bottlenecks of an OS running a specific workload, but as a general purpose technique it has limited applicability because it requires often-unavailable source code for the OS kernel. This leaves microbenchmarks.

Until recently, the only widely-available OS microbenchmark suite was a set of tests developed by Ousterhout to measure the performance of the Sprite operating system [12]. Ousterhout's benchmarks isolate a number of kernel primitives and, when run across multiple platforms, provide some indication of the dependence of OS performance on machine architecture. However, they do not include enough detailed tests to characterize the capability of the underlying hardware and to use that characterization to understand the performance of higher-level kernel primitives. In 1995, McVoy improved the microbenchmark state-of-the-art with the introduction of his *lmbench* package: a suite based on a broad array of portable OS microbenchmarks capable of measuring both hardware capabilities (e.g., memory bandwidth and latency) and OS primitives (e.g., process creation and cached file reread) [11]. Through its detailed tests, *lmbench* offered the possibility of decomposing operating system primitives into their hardware-dependent components, thus providing the tools necessary to develop an understanding of the relationship between architectural features and operating system performance.

On its own, however, *lmbench* is just a set of tools; for us to use it as the basis for understanding operating system performance, we needed to develop a benchmarking methodology: a means of using the tests and their results to build a decomposition of performance from high-level primitives down to hardware details. To see if this was possible, we undertook an in-depth study of the performance of the NetBSD operating system running on the Intel x86 architecture. We chose NetBSD for its openness and its multi-platform support: having the source code meant that we could use kernel profiling to verify our techniques, and its multi-platform support provided the possibility of future cross-architecture comparisons. We selected the Intel x86 architecture as our subject architecture due to its breadth: in its evolution from the i386 through the Pentium Pro, the Intel x86 architecture has progressively included more and more of the advanced features that characterize a modern architecture, including pipelining, superscalar execution, and an out-of-order core with an integrated second-level cache.

As we used *lmbench* to probe into the details of NetBSD's performance and its interaction with the x86 architectural features, we found that it had several shortcomings as a tool for the detailed scientific study of OS-hardware interaction. Most notably, it did not provide the statistical rigor and self-consistency needed for detailed architectural studies. To resolve these shortcomings, we decided to revise *lmbench* into a suite of tests that would be useful for both cross-platform comparison and detailed system analysis—we wanted to fulfill the *lmbench* promise of providing a set of tools ca-

This research was supported by Sun Microsystems Laboratories.

pable of illuminating the inner workings of an operating system in order to bring to light how that operating system's performance depends on the hardware upon which it runs. Since *lmbench* provides a sufficiently complete set of tests to cover a broad range of operating system functionality, our modification efforts were directed at making the existing tests more rigorous, self-consistent, reproducible, and conducive to statistical analysis.

In this paper we first detail the modifications that we made to *lmbench*, then proceed to demonstrate how we applied our new benchmark suite, "*hbench:OS*," to the problem of understanding, at the most detailed levels, the dependence of OS performance on architectural features of the Intel x86 platform.

The rest of this paper is organized as follows. In Section 2, we discuss the revisions we made to *lmbench*. Section 3 describes the methodology that we developed for using *hbench:OS* to analyze system performance. Section 4 presents the NetBSD case study, and we conclude in Section 5.

2 Revising *lmbench*

As we began to apply *lmbench* to our analysis of NetBSD/x86 performance, we encountered both minor problems (we found the output format of the benchmarks difficult to analyze) and more substantial problems (with a reasonable compiler, the test designed to read and touch data from the file system buffer cache never actually touched the data). Our biggest concerns, however, were with the benchmarks' measurement and analysis techniques: we were not confident that the methodology used in a number of tests was rigorous enough to produce accurate, reproducible results. In the following sections we document the difficulties that we encountered and the methods that we used to solve them. The original *lmbench* tests that are used in this paper are summarized in Table 1; we refer the reader to McVoy's original *lmbench* paper [11] for a more detailed description of the benchmark tests discussed.

For the remainder of this paper, we use *lmbench* to refer to the original *lmbench–1.1* test suite, and *hbench:OS* to refer to the modified test suite. Also, we will refer to the on-chip cache as the L1 (first level) cache and the secondary cache as the L2 (second level) cache. Note that the Pentium Pro integrates the L2 cache into the same package as the CPU, while earlier CPUs use an external L2 cache.

Test	Description
Memory read/write bandwidths	Determines the bandwidth to memory by timing repeated summing of a large array.
bcopy() bandwidth	Determines the memory bandwidth achieved by the bcopy() memory copy routine.
File reread bandwidth	Measures the bandwidth attainable in reading cached files from the system buffer cache.
TCP bandwidth	Measures the bandwidth attainable through an already-established TCP connection through the loopback interface.
Cached mmap-read bandwidth	Measures the bandwidth attainable when reading from a cached file mapped into the process's address space
Process creation	Measures the latencies of three different methods of process creation: via a simple fork(), fork()+exec(), and system().
Signal handler installation	Measures the latency of installing a new signal handler from a user process.
TCP connection latency	Measures the latency of setting up a TCP connection across the loopback interface.

Table 1: Summary of a subset of the original *lmbench* benchmarks [11].

2.1 Timing Methodology

Lmbench performs all of its timing using the `gettimeofday()` system call to sample the system time before and after the operation that is being measured. On systems that do not have (or use) hardware microsecond timers, the resolution of `gettimeofday()` is only that of the system clock—as coarse as 10 ms in some cases. One particularly severe real-life example that demonstrates the problems imposed by a coarse-grained timer can be seen in the DEC Alpha 21164 running Digital UNIX 3.2F; the resolution of `gettimeofday()` on such systems is 1 ms. This is far too coarse to accurately time individual low-latency events or to measure high bandwidths, as some of the *lmbench* tests attempt to do. For example, the *lmbench* TCP connection latency benchmark times individual connection requests through the loopback interface; as these take much less than 1 ms, *lmbench* reports a 0 microsecond connection latency on the Alpha. Similarly, the memory bandwidth benchmark times a single buffer read; if the test is run with buffers small enough to fit in the L1 or L2 cache, *lmbench* on the Alpha reports infinite bandwidth.

We made two modifications to avoid the timer resolution constraint imposed by `gettimeofday()`. First, we modified each benchmark to run its tests in an internal loop, timing the entire loop and reporting the average time (total time divided by number of iterations). While many of the *lmbench* latency tests already used such internal loops, the loops were run an arbitrary, predetermined number of times, causing scalability problems on different-speed systems. To fix this, we modified the internal loops to run for a minimum of one second, calculating the number of iterations dynamically. The dynamic calculation of the iteration count ensures that the running time of the benchmark will exceed any reasonable timer resolution by a factor of 10 or more, regardless of the system or CPU being used. In addition, the inclusion of internal iteration with the bandwidth tests makes possible precise measurement of the memory and copy bandwidths to the L1 and L2 caches.

For benchmarks where the measurement is destructive and can only be taken once (for example measuring the virtual memory and TLB overhead in reading a memory mapped region), the loop-and-average method is not effective. For these tests, we had to appeal to a hardware-specific solution to gain the timing accuracy needed: we introduced hooks to allow hardware cycle counters (which tick at the CPU's internal clock speed) to replace `gettimeofday()` for timing. Currently *hbench:OS* only supports the Pentium and Pentium Pro counters, but adding support for other architectures (such as the Alpha or SuperSPARC) is not difficult. Note, however, that if an architecture supports no hardware counters/timers, it is not possible to measure such destructive events accurately.

Adding the hardware-timer hooks also significantly enhances the flexibility of the *hbench:OS* package, as the high-resolution timers give *hbench:OS* the capability of measuring events with low latencies without the need to run the event in a loop, thus allowing collection of cold-cache performance numbers. When using the `gettimeofday()` timing method, only warm-cache results can be measured, as the loops that are required for accuracy also allow the benchmark to run entirely from the cache.

Our last modification to the timing routines was to include code to measure and remove the overhead introduced by the timing mechanism (either the `gettimeofday()` system call or the instructions to read the hardware counters). Removal of this timer overhead is essential, especially when using the hardware timers to measure single low-latency events. When combined with the use of the hardware counters, this allows for precise timing measurements: on a 120 MHz Pentium, for example, our timings are accurate to within one clock cycle, or 8.3 ns.

2.2 Statistical Methodology

With the timing irregularities solved through iteration and the use of hardware counters, we discovered another shortcoming in *lmbench*'s methodology: it was inconsistent in its statistical treatment of the data. Several of the benchmarks reported the result of one measurement, others reported the average of multiple repeated measurements, and yet others reported the minimum of multiple repeated measurements. We wanted to run each test a number of times to obtain more statistically sound results, but with the goal of applying a consistent policy to the data analysis. To achieve this goal, the benchmarks were each restructured to make a single timing measurement. The tests are run multiple times by a driver script (each run in a new process), and the result from each run is appended to a file. Since our reformulation of the tests preserves the value from each run of the benchmark, we have divorced the data analysis policy from the benchmark itself.

The most-used statistical policy in *lmbench* is to take the minimum of a few repetitions of the measurement; this is intended to pick out the best possible result by ignoring results contaminated by system overhead. However, in doing so it can pick out results that are flukes—especially when the measurement involves subtracting an overhead value, as in the context-switch latency benchmark. If the actual overhead on a specific run is lower than the pre-calculated overhead, an abnormally good result will be obtained when the pre-calculated overhead is removed from the result. To avoid these problems, in most cases we take an *n*%-trimmed mean of the results: we sort the results from a benchmark, discard the best and worst *n*% of the values, and average the remaining (100–2*n*)%. *n* is typically 10%. With this policy, we discard both the worst values resulting from extraneous system overhead as well as the overly-optimistic results.

For certain benchmark tests, however, a simple trimmed mean is not sufficient to capture all of the important features of the results. This is particularly noticeable when the results of a test do not approximate a normal distribution, but are (for example) bimodal. Such cases are easily detected by their large standard deviations, and since all data is preserved, it is easy to view the actual distribution of the data to determine the best interpretation. We encountered this problem in measuring L2 cache bandwidth, as cache conflicts within our test buffer produced a bimodal distribution where the true bandwidth was represented by a large, narrow peak and the false (conflict) bandwidth was represented by a lower peak with larger spread. In this case, we merely increased the percentage that was trimmed from the data in order to isolate only the true bandwidth peak.

Finally, we have modified the benchmarks (where possible) to perform one iteration of the test before beginning the real measurement. Since we run most of the tests in loops anyway, we expect warm-cache results. Running the test once before commencing measurement ensures that the caches are primed and that any needed data (e.g., files in the buffer cache) are available.

Note that in gathering the results in this paper, we ran each benchmark (each of which runs a large number of internal iterations) fifty times on all machines but the *386-33* and *486-33* (due to limited access to the hardware, only five iterations were performed on these machines), and we report the 10% trimmed average across these iterations. Standard deviations are represented by error bars in the graphs; in all cases standard deviations were less than 1% (and are frequently not visible in the graphs) except in the file reread benchmark, which produced standard deviations of less than 5%.

2.3 Increased Parameterization

In order to make the *lmbench* tests more amenable to our investigations, we made several modifications to increase the flexibility of the benchmarks by making them more parameterizable. For example, we modified the pipe, TCP, and file-reread bandwidth tests to accept a transfer size as an argument in order to investigate the effect of write-back caches on small-buffer transfers. We also modified the memory read/write/copy bandwidth tests to allow for measurement of the L1 and L2 cache bandwidths. Finally, we modified the process creation benchmark to allow for measurement of both dynamic and statically-linked processes.

2.4 Context Switch Latency

Measuring context switch latency is particularly challenging, as the latency of a context switch is not very well-defined. In the strictest sense, context switch latency is the time that it takes for the OS to suspend and save the hardware state of a running process (e.g., registers, stack pointer, page table pointers), select a new process to run, load the new process's saved hardware state, and then begin executing it. *Lmbench* uses a looser definition of context switch latency: in addition to the above components of context switch time, it includes the latency that results from faulting the working set of the new process into the CPU's cache. This cache-filling overhead is not strictly a part of context-switch time, for it only occurs when the two processes collide in the cache; thus, it is a function of the sizes of the processes' working sets and the OS's page-mapping policy, and not of the hardware or of the OS's context-switch code. What *lmbench* measures is closer to what a user might see for context-switch time with several large, data-intensive processes than to the raw context-switch speed of the OS.

Although measuring cache conflict overhead is useful (especially for estimating context-switch time for large processes), *lmbench*'s context switch benchmark demonstrates that there are problems with this approach that make it infeasible for a portable context switch benchmark. The most significant problem occurs when the operating system does not support intelligent page coloring, i.e., it chooses physical addresses for virtual pages randomly[1]. To understand why this is a problem for *lmbench*, we need to investigate how *lmbench* collects its context switch latency data.

The *lmbench* context switch latency benchmark measures the time to pass a token around a ring of processes via pipes; to duplicate the effect of a large working set, each process sums a large, private data array before forwarding the token, thereby forcing the pages of the array into the cache. When the total time for this operation is divided by the number of processes in the ring, *lmbench* is left with a number that includes the raw context-switch time, the time to fault the array into the cache, the time to sum an already-cached array, and the time to pass a token through a pipe. The latter two factors are measurement overhead and must be removed. To do so, *lmbench* passes a token through a ring of 20 pipes within one process, summing the same data array each time the token changes pipes, then divides by the number of times the token went through a pipe. The problem with this approach is that the test assumes that summing the buffer produces no unnecessary cache conflicts, for the summing overhead should not include any cache-fill time. However, if the virtually-contiguous pages of the buffer are randomly assigned to physical addresses, as they are in many systems, including NetBSD, then there is a good probability that pages of the buffer will conflict in the cache, even when the size of the buffer is smaller than the size of the caches [3]. Thus the overhead will contain some cache-fill time, and as a result might be too high; if the actual context switch test obtains good page mappings, the overhead may even be so high that when *lmbench* subtracts it from the total time to get just the context-switch latency, the resulting (reported) context switch latency is negative or zero. A similar problem exists if the overhead-measurement test obtains good page

1. It is also this random page-mapping policy that introduces the somewhat large (5%) standard deviations that we see in the file reread benchmark.

mappings while the real context switch latency test obtains conflicting mappings; here the overhead will be too small, and the reported context switch latency will be too large.

Because with *lmbench* there is no guarantee of reproducible context-switch latency results in the absence of OS support for intelligent page coloring, we decided, in *hbench:OS*, to restrict the test to measure only the true context switch time, without including the cost to satisfy extra cache conflicts or to fault in the processes' data regions, as these can be approximated from the cache and memory read bandwidths. To this end, we introduced a new context switch latency benchmark to supplement the existing *lmbench* test. We did not replace the *lmbench* test completely, as it can be useful in estimating user-visible context-switch latencies for applications with a known memory footprint, and for determining cache associativity. In our new test, context switch latency is a function of the speed of the OS in selecting a new process and changing the hardware state to run it. To accomplish this, we carve each process's data array out of a large region shared between all the processes in the ring. To compute the overhead for *nproc* processes, we measure the time to pass a token through *nproc* pipes in one process, summing the appropriate piece of the shared region as the token passes through each pipe. Thus we duplicate exactly what the real context switch test does: we use the same memory buffers with the same cache mappings, and touch them in the same order. When we subtract this overhead from the context switch measurement, we are left with the true context switch time plus any hardware-imposed overhead (such as refilling non-tagged TLBs and any cached data that got flushed as a result of the context switch but not as a result of faulting in the process). With these modifications, we can obtain results with a standard deviation of about 3% over 10 runs, even with large processes, and without having to flush the caches. In contrast, on the same machine, *lmbench* reports results with standard deviations greater than 10%.

2.5 Memory Bandwidths

In the interest of consistency, we made some modifications to the benchmarks that touch, read, or write memory buffers. The *lmbench* bandwidth tests use inconsistent methods of accessing memory, making it hard to directly compare the results of, say memory read bandwidth with memory write bandwidth, or file reread bandwidth with memory copy bandwidth. The tests that read memory primarily use array-offset addressing to iterate through the buffer, while the write and copy-based benchmarks dereference and increment pointers. On pipelined or superscalar architectures, using array-offset addressing produces address generation interlocks (due to the implicit add), while using pointers can cause false data dependency interlocks. The difference between the two approaches is evident upon examination of the compiler's output for the two benchmarks: gcc (on the x86) implements the array-offset addressing in the C statements (`ebx[0]=1; ebx[1]=1;`) as:

```
movl $1, (%ebx)
movl $1, 4(%ebx),
```

while a similar example using pointers (`*ebx++ = 1; *ebx++ = 1;`) is implemented as:

```
movl $1, (%ebx)
addl $4, %ebx
movl $1, (%ebx)
addl $4, %ebx.
```

Depending on how the processor's pipeline handles interlocks, the two methods can produce different timings. For example, on the Alpha processor, memory read bandwidth via array indexing is 26% faster than via pointer indirection; the Pentium Pro is 67% faster when reading with array indexing, and an unpipelined i386 is about 10% faster when writing with array indexing. To avoid errors in interpretation caused by these discrepancies, we converted all data references to use array-offset addressing. In addition, we modified the memory copy bandwidth to use the same size data types as the memory read and write benchmark (which use the machine's native word size); originally, on 32-bit machines, the copy benchmark used 64-bit types whereas the memory read/write bandwidth tests used 32-bit types.

3 *hbench:OS* as a Scientific Tool: Philosophy and Methodology

The previous section described the means by which we were able to convert *lmbench* into a more rigorous analytical tool. Before proceeding to describe how we used our new tool for detailed measurements of several x86 systems, let us pause for a moment to consider why such a tool is useful. Much of system measurement today is done at only one point on the continuum of abstraction that stretches from the silicon of the hardware through the operating system and up to the interface presented by user applications. However, it is important to try to characterize an entire system, from hardware features to application performance. *hbench:OS* is a first step toward this ultimate goal: it allows for the characterization and analysis of the part of the performance continuum reaching from the hardware to the performance of the OS and OS-dependent applications, and provides a foundation for future work to extend the analysis to the level of user applications.

However, *hbench:OS* used in a vacuum will not provide the information needed to understand the interactions and performance characteristics of a given system. Thus we next propose a methodology for analyzing and interpreting its results. The benchmarks in *hbench:OS* can be roughly divided into two layers: one that quantifies hardware capabilities (such as memory bandwidth), and another that measures the primitive functionality that is exported from the kernel to applications (such as system calls, process creation, and file/network access). When these layers are combined with the hardware at the bottom and OS performance at the top, a hierarchical structure emerges. Figure 1 depicts this hierarchical structure as a pyramid of relationships between layers representing components of OS performance. Our goal in creating a methodology for *hbench:OS* was to provide a means of reconstructing the interdependencies in the pyramid. Thus, when the methodology is applied, the result is a decomposition of the performance of the highest-level kernel primitives (those seen by kernel-dependent applications) into the performance of the underlying hardware; such a decomposition can then be used to predict which architectural features influence OS performance.

Our methodology consists of two steps: first, using *hbench:OS* to measure performance at each level of the pyramid while varying features of the hardware; and second, using the changes in hardware as well as software analysis (via hardware counters, software profiling, or code analysis) to relate the performance of primitives in a given layer to the performance of layers above and below it. Once the interaction between each pair of adjacent layers is understood, the pyramid of performance dependencies can be reconstructed.

In many cases, the *hbench:OS* tests provide enough detail about the individual layers of the pyramid so that such a reconstruction is possible: both bulk data transfer primitives and process creation primitives can be decomposed using the pyramidal model. In the other cases, where *hbench:OS* does not provide detailed benchmarks to quantify the hardware capabilities used by a higher-level primitive, it is necessary to bypass the middle layer of the pyramid and determine directly the hardware dependence of each test. These results can be gleaned from the information obtained by analyzing how the results of a particular benchmark change when the hardware is varied in a controlled manner (i.e., when only one component of the system is changed at a time). We found this technique

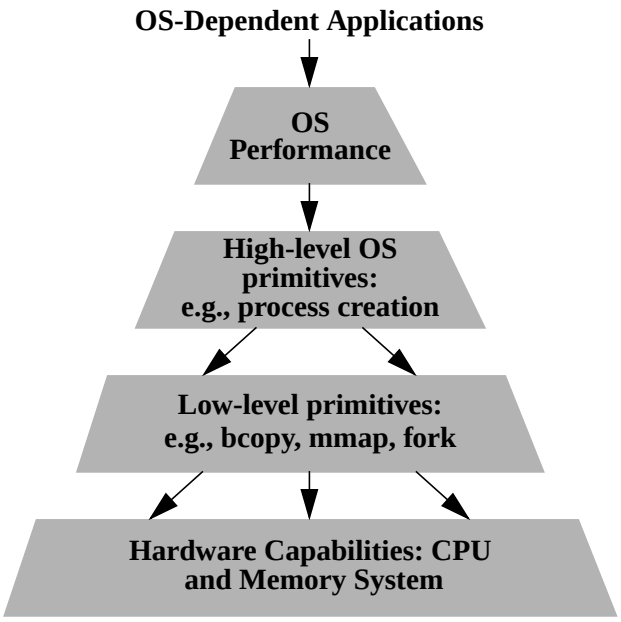

Figure 1: Decomposition of OS performance via *hbench:OS* primitives. The performance of OS-dependent applications (such as web servers) can be decomposed into high-level OS-provided services and primitives, which can in turn be decomposed into low-level kernel primitives, which can again be decomposed into hardware capabilities. In many cases, *hbench:OS*'s suite of tests allows us to measure and relate the lower three levels of this hier-

especially useful for relating the lowest-level *hbench:OS* primitives (such as memory read bandwidth) to features of the hardware architecture.

4 An *hbench:OS* Case Study: The Performance of NetBSD on the Intel x86 Platform

With both our new benchmark suite and a methodology for using it in hand, we returned to our original task of studying the architectural basis of OS performance on the Intel x86 architecture. For our subject OS, we chose NetBSD 1.1, a derivative of the CSRG 4.4BSD-Lite release [5], which shares a common ancestry with many of today's commercial UNIX implementations. For our systems, we selected eight machines: a 386, two 486's, four Pentiums, and one Pentium Pro. The hardware details of these machines are given in Table 2. All of our machines ran the same NetBSD-1.1PL1 "GENERIC" kernel; we did not optimize the kernels for their target platforms, for we were particularly interested in the effects of hardware evolution on operating system performance in the absence of processor-specific optimizations.[2]

Comparisons between benchmark results from various subsets of our test machines reveal dependencies on features of the CPU architecture and the memory system. For example, comparing the 100 MHz Endeavor Pentium with the 100 MHz Premiere-II Pentium reveals the effect of pipelining the L2 cache and installing EDO memory; similarly, comparisons between the 90, 100, and 120 MHz Endeavor Pentiums reveal the effects of increasing the CPU clock rate while holding the memory system constant. The

2. This issue is especially important for portable OS's that may not be tuned for a particular architecture (e.g., Linux, Windows NT, UNIX), as well as for OS software that can reasonably be expected to outlive the hardware for which it was originally optimized (e.g., Windows 3.x).

Name-MHz	Caches Features	Memory/ Bus-MHz	Processor
386-33	no L1 64K async. L2	70 ns 33 MHz	i386DX
486-33	8K combined L1 256K async. L2	60 ns 33 MHz	i486DX
486-66		60 ns 33 MHz	i486DX2
Endeav-90	16K split L1 512K pipeline-burst L2	60 ns EDO 60 MHz	Pentium (i430FX chipset)
Endeav-100		60 ns EDO 66 MHz	Pentium (i430FX chipset)
Endeav-120		60 ns EDO 60 MHz	Pentium (i430FX chipset)
Prem-100	16K split L1 512K async. L2	70 ns 66 MHz	Pentium (i430NX chipset)
Pro-200	16K L1, 256K L2, both write-back and on-chip	60 ns EDO 66 MHz	Pentium Pro (i440FX chipset)

Table 2: Features of Test Machines. Note that the 100 Mhz Pentiums run the memory bus at 66 MHz as opposed to the 60 MHz of the other Pentium processors. Unless otherwise noted, all L1 caches are write-through.

specific comparisons that we used and the conclusions that we drew are detailed in the following sections.

4.1 Bulk Data Transfer

We begin our study with an example of the primary methodology discussed above, the decomposition of application-level OS primitives all the way down to hardware capabilities. The most illuminating example of this is the case of bulk data transfer. We choose bulk data transfer as an illustrative metric since it is an essential component of the performance of bandwidth-sensitive applications such as web servers and multimedia/network video applications. When running a heavily-used web server, bcopy is the most-frequently called kernel function, accumulating more than 21% of the total in-kernel time. Even typical development work involves large amounts of bulk data transfer: our kernel profiling results under NetBSD indicate that the kernel can spend as much as 23% of its time in bcopy while supporting a mix of editing, compiling, debugging, and mail.

Applications that rely on bulk data transfer use one of three methods to access their data: reading from a file in the file system, sending and receiving data on a TCP connection, or mapping a file into their address space. Since each of these data-access methods involves a significant number of memory accesses, we can base our decomposition on the *hbench:OS* tests that measure the hardware memory read, write, and copy bandwidths. If we ignore the effects of disk and network latency (since we run all of our disk tests within the buffer cache and all of our network tests on the loopback interface), we arrive at the decomposition shown in Figure 2. There is also a CPU computation component in each of the application-level primitives; it is most significant in the TCP test due to the complexity of protocol encapsulation and checksumming.

Hardware Bandwidth Capabilities

The hardware's ability to move data is a function of the main memory speed, the memory bus bandwidth, the size of the L1 and L2 caches, the write policy of the caches (e.g., write-back, write-through, write-allocate), and the processor's ability to efficiently use these resources (i.e., via pipelining or reordering memory operations). It is not possible to directly measure any one of these

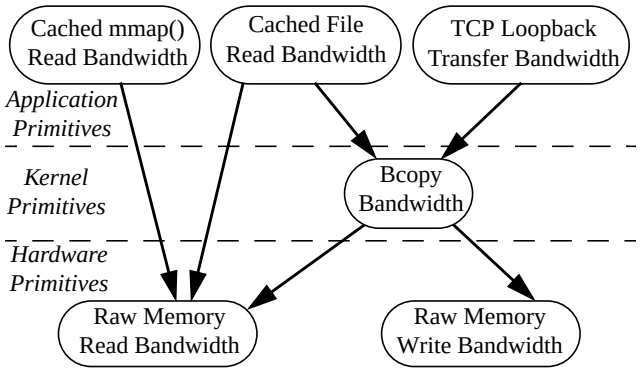

Figure 2: Decomposition of Application Data-access Primitives. All of the application-level data primitives for bulk data access depend on the hardware's memory read bandwidth since they all touch data. File reading interposes the extra overhead of a cross-protection-domain bcopy to move the requested data to user-space buffers; TCP transfer interposes three bcopy's as it shuffles the data through the loopback interface and between user and kernel space.

features; *hbench:OS* measures the interaction of all the components of a particular system. However, by using comparisons between different system configurations, we can measure how each component affects performance.

The *hbench:OS* tests that can be used to quantify the hardware's capability for bulk data transfer, i.e., those which measure the bottom layer of Figure 2, are the raw memory bandwidth tests, which measure effective software read and write bandwidths—the attainable bandwidths when array-addressing operations (needed to index through memory) are inserted between each memory reference. Although the raw hardware transfer bandwidths are potentially higher, the software bandwidths are more representative of what is attainable by actual code.

Figure 3 plots the peak raw bandwidths for reading from and writing to both caches and main memory of several of the test machines. The almost 4-fold improvement in L2 and main memory read performance between the *486-66* and the *Prem-100* is due to increased bus bandwidth and bus burst capability. The Pentium system has a 64-bit data bus, twice as wide as the 486's 32-bit bus; in addition, the Pentium supports burst transfers from the system's fast page mode DRAM, while the 486 does not. The measured write performance only doubles from the *486-66* to the *Prem-100*, because the older chipset on the Premiere system does not burst writes to DRAM; only the wider path to memory plays a role in the speedup compared to the 486.

The write performance of the *Endeav-100* doubles that of the *Prem-100* because of the Endeavor motherboard's pipeline-burst L2 cache and EDO DRAM. The pipeline-burst cache can latch three out of every four memory references in one bus cycle each and then burst them off to the DRAM. This explains why the main memory write bandwidth is comparable to the L2 cache's inherent read bandwidth—the pipelined cache is hiding much of the already-low DRAM latency from the CPU. Note that on the *Endeav-120*, which shares the same memory subsystem as the *Endeav-100* and *Endeav-90*, the DRAM and L2 read bandwidths are higher than expected from comparison with the *Endeav-90*, since the processor is clocked at an integral multiple of the memory bus speed. This allows the *Endeav-120* to utilize more of the bus bandwidth (61% vs. 54% for the *Endeav-100*, as determined with the Pentium hardware event counters) since CPU and bus cycles coincide more frequently.

It is interesting to note that in the raw memory bandwidth tests, the dual-issue capability of the Pentium is being very poorly utilized. We instrumented *hbench:OS* with the Pentium's built-in hardware counters, and discovered that when memory is accessed by summing an array using array-offset instructions, less than 0.1% of the memory instructions are parallelized. Similar results are found when the built-in string opcodes are used. Parallelism can be increased to nearly 50% by using pointer arithmetic to step through the array. In this case, each pointer increment is issued along with a memory reference, and is essentially free; however, two memory references are never issued simultaneously. In addition, this extra parallelism is introduced at the cost of an extra stall cycle on each memory access due to address generation interlocks. Thus both methods of memory access provide approximately the same performance, so we predict that memory intensive workloads may profit less than expected from the superscalar architecture of the Pentium.

This conclusion also raises the interesting issue of the usefulness of micro-optimizing compilers for the OS kernel. We experimented with the PCG version of pgcc (an adaptation of the GNU gcc compiler that performs aggressive instruction scheduling for the Pentium pipelines) and discovered that pgcc's optimizations had essentially no effect on the performance of the memory-intensive benchmarks, even when the memory accesses were explicitly coded (as opposed to using the built-in string operations). The problem is that the hardware itself does not allow dual-issue of memory references in the cases we tested, and thus no instruction scheduling policy could improve performance in these cases.

Returning to the data in Figure 3, we see that the most spectacular feature is the performance of the Pentium Pro system. The *Pro-200* exhibits a strange combination of impressive across-the-board memory bandwidth, except for uncharacteristically poor main memory write bandwidth. The *Pro-200*'s nonblocking write-back L1 cache gives it an extreme performance advantage over the Pentiums on small cached reads and writes. The *Pro-200* L2 cache also

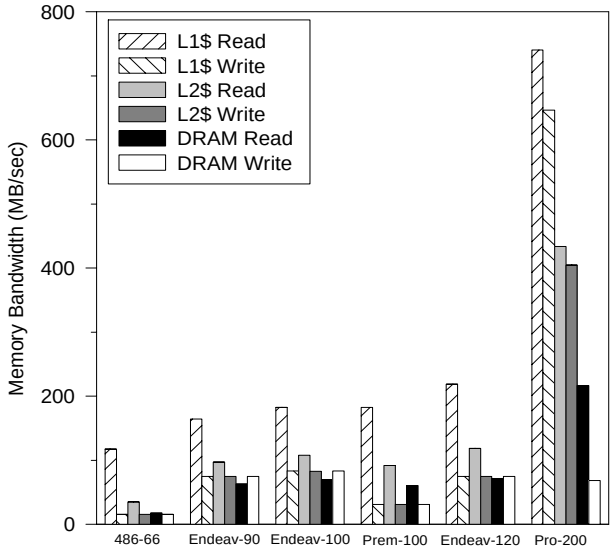

Figure 3: Raw Memory Bandwidth. The 64-bit, burst-capable memory bus of the Pentiums produces a factor of four improvement in L2 and DRAM read memory bandwidth from the i486 architecture. The combination of pipeline-burst cache and EDO DRAM gives the *Endeav-100* a significant performance advantage over the *Prem-100*; its higher memory bus clock allows the *Endeav-100* to outperform its 90 and 120 MHz siblings. The Pentium Pro exhibits exceptional cache performance and good memory read bandwidth (due to its out-of order prefetching memory unit), but suffers on memory writes due to an unnecessary cache coherency protocol that prevents back-to-back bus write transactions.

significantly outperforms that of the Pentiums, as the Pentium Pro runs its on-chip, lockup-free L2 cache at the CPU clock speed, as opposed to the system bus speed. Also, while the Pentiums' non-write-back caches access memory on every write, the Pentium Pro's write-back caches are intelligent enough to combine writes into cache-line-sized increments, resulting in cached write performance that nearly equals cached read performance, as the write-back cache is not forced to read a line before writing to it. The astounding cache performance on the *Pro-200* suggests that write-back caches offer a major performance advantage to those applications that perform bulk data transfer in small, cache-sized chunks, for example, the size of a typical HTML file.

Along with its high cache performance, the *Pro-200* also sports exceptionally high main memory read bandwidth. In fact, the 216 MB/s that it achieves approaches the 226 MB/s theoretical maximum bandwidth out of 3/2/2/2-clocked EDO DRAM on a 66 MHz bus. The reason for this exceptional performance is twofold. First, the Pentium Pro sports an out-of-order execution engine that is capable of reordering memory reads and removing the data dependencies implicit in the benchmark. By using register renaming and speculative memory reads, the Pentium Pro can implicitly batch and prefetch data reads, thus allowing it to issue memory reads as fast as the external memory system can handle them. Second, and more importantly, the Pentium Pro's pipelined, transaction-based system bus allows it to issue consecutive back-to-back data read transactions without incurring bus turn-around time and transaction set-up costs [8]. In contrast, the Pentium executes all memory operations in sequence, inserts extra data dependency stalls due to its small register set, and negotiates for the system bus on each read request.

The *Pro-200*'s main memory write bandwidth, in contrast, is exceptionally low—almost 18% slower than the write bandwidth of the *Endeav-100*, a system with identical DRAM and the same bus speed. To determine why this was the case, we instrumented the benchmark with the Pentium Pro's built-in hardware counters [9]. For each 32-byte line of data written by the CPU, the counters indicate that two bus transactions take place: a writeback transaction and a read-for-ownership (RFO) transaction. The writeback is expected, since as the CPU stores a line into the cache it must displace an existing dirty line from a previous write. The RFO on the line about to be written is used to guarantee cache-coherency: the CPU must ensure that no other CPU in the system has a dirty copy of the line it is about to write. However, there is no need for a read-for-ownership transaction in our case, as the *Pro-200* is a single-processor system, and thus there are no other CPUs that could contain a dirty line; there is similarly no need to read the entire line, as we have seen in the L2 cache bandwidth that the write-back cache is intelligent enough not to load a line that is about to be entirely rewritten. Thus by interspersing a RFO transaction between each write, the available bus bandwidth drops significantly, as the CPU must renegotiate for the bus on each write, instead of performing back-to-back writes (as it does in the read case). Also, there is the bus overhead of the read-for-ownership transaction itself, and the bus turn-around time needed to switch between the transactions. Thus it seems that requiring the demonstrably high overhead of a RFO-based cache coherency protocol even when there is only one CPU in the system is a suboptimal design, as it severely cripples the available memory write bandwidth on the Pentium Pro.

It appears that Intel may have attempted to compensate for this design by including an undocumented "FastStrings" flag in one of the Pentium Pro's control registers: when FastStrings are enabled, the RFO transactions are converted to Invalidate transactions (so the cache does not read the new line but merely invalidates it in other CPUs). However, on a single-CPU system the Invalidate transaction is still unnecessary since there is only one cache on the bus. Additionally, this feature only improves DRAM write bandwidth slightly (about 5%) and only when certain string instructions are used to perform the write; converting the RFOs to Invalidates does not remove the bus transaction and renegotiation overhead, the major factor in the low DRAM write bandwidth.

Kernel and Application Bandwidth Primitives

From the *hbench:OS* measurements of the hardware capabilities of each machine, we can now generalize to the kernel primitive, *bcopy*, and from there to the application primitives such as file reread and TCP throughput. If each primitive were completely dependent on the memory subsystem, we would expect to see similar patterns as were discovered with the hardware primitives; any deviation from these hardware patterns should indicate that the primitive showing the deviation has a non-memory-system dependent component.

The primary kernel primitive relied upon by bulk-data application primitives is *bcopy*, used to transfer data around the kernel and between kernel and user space. Our bcopy benchmark uses the libc bcopy routine (identical to kernel bcopy in NetBSD) to copy both cached and uncached buffers in user space; this routine uses the x86 string instructions to efficiently move data. In the ideal case, we expect the results of the bcopy benchmark be one-half of the harmonic mean of the read and write bandwidths for each machine, since each byte copied requires one read and one write. However, when reads and writes are combined into copies, unexpected interactions can develop and cause the measured copy bandwidth to exceed or fall short of the half-harmonic mean prediction. These are the more interesting cases, as they illustrate optimizations or flaws in the hardware design, and how such design characteristics affect performance. In Figure 4, we present the results of the non-cached bcopy test along with the half-harmonic means calculated from the

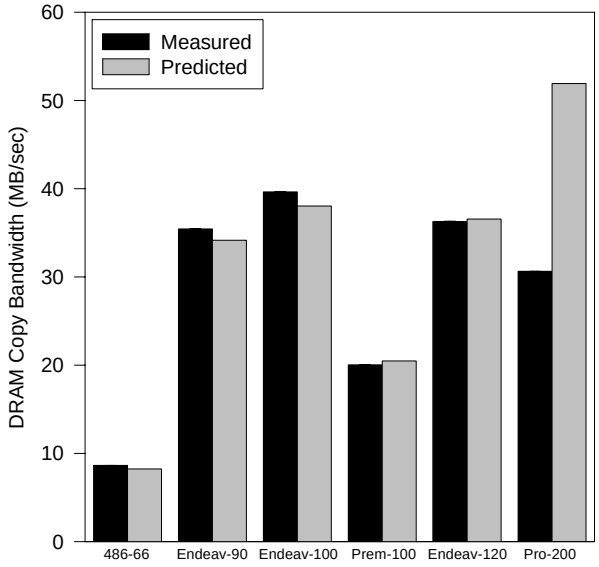

Figure 4: bcopy Bandwidth (2MB buffers). The memory systems determine performance on this benchmark: the predicted bcopy results (one-half of the harmonic mean of the read and write memory bandwidths) track closely with the measured numbers. When the 100 MHz Pentium is scaled to account for its higher bus clock rate, all the Endeavor-based Pentium systems achieve identical bcopy bandwidths, independent of processor speed. The *Prem-100*, with a slow memory system, attains only half the bandwidth of the identical processor with a newer memory system (*Endeav-100*). The *Pro-200*'s dismal memory write bandwidth leads to unexceptional bcopy performance; the actual performance falls far short of the predicted performance because of bus turnaround time not accounted for in the read bandwidth.

raw bandwidth results in Figure 3; the cached bcopy results are similar. For all systems but the *Pro-200*, the graph shows the expected result that bcopy bandwidth is directly correlated with the raw memory bandwidths. The measured results slightly exceed the predictions in most cases because the CPU (executing the x86 string operations) can issue the reads and writes back-to-back, without decoding and executing explicit load and store instructions.

Those machines with poor raw write bandwidth suffer in the bcopy test, since both read and write bandwidths have an equal influence on the copy bandwidth: for example, although it uses the same processor, the *Prem-100* achieves only half the bcopy bandwidth of the *Endeav-100*. This again demonstrates the effectiveness of an enhanced memory subsystem with a pipelined L2 cache and EDO DRAM at improving performance of operations requiring the movement of large quantities of data. The disappointing write performance of the Pentium Pro memory system on large buffers completely negates the advantages of its advanced cache system, resulting in bcopy performance that is actually worse than that of some of the Pentiums. The *Pro-200*'s copy bandwidth also falls far short of our prediction, since, when copying, the processor cannot issue back-to-back reads on the system bus, and must alternate read, write, and read-for-ownership transactions; each new transaction requires setup and negotiation overhead. Again, enabling Fast-Strings on the Pentium Pro has little effect (less than 1%) because the extra coherency transaction is still present.

With this understanding of bcopy, we move on to consider the application-level data-manipulation primitives: cached file read, local TCP data transfer, and mmap()'d file read. We will only consider the first two methods here; full details on mmap()'d file read can be found in a more detailed report on this system comparison [2]. From the decomposition presented earlier in Figure 2, we expect that a significant component of the attainable bandwidths for each of these primitives is due to a dependence on the memory system, and thus we expect that the architectural changes that have enhanced memory system performance (such as faster, wider busses and pipelined caches) will enhance the performance of these primitives as well. We now examine each of these three bandwidth measurements in order to determine if this is the case, or if other factors besides the memory system are involved.

The *hbench:OS* file reread benchmark measures the bandwidth attainable by an application reading data from a file present in the kernel's buffer cache; we used 64KB read requests for this test. For each byte transferred in this test, NetBSD performs one memory read from the kernel's buffer cache, one memory write to the user buffer, and a final memory read as the benchmark touches the byte. This is one more memory read than the bcopy test, so one might expect file reread to be significantly slower than bcopy. Similarly, the TCP bandwidth test involves transferring 1MB in-memory buffers over the local loopback interface. In this test, each byte transferred must be copied three times, so we expect at least a 3-fold performance degradation relative to bcopy.

The results for these two benchmarks on several of our test machines are shown in Figure 5 along with predicted results derived from the bcopy test and raw bandwidth tests. The TCP bandwidths show the expected pattern: the relative performance is comparable to that of bcopy, while the magnitude is approximately one-third that of bcopy. As expected, there is a partial CPU dependency, since TCP's checksumming and encapsulation require more processing than bcopy; the Pentium Pro's out-of-order execution allows it to overlap some of the computation and memory references involved in the TCP processing, giving it a slight performance edge. However, it is clear that the memory system still dominates TCP transfer bandwidth.

The file reread results also show similar relative performance to the bcopy results, with the exception of the Pentium Pro. Although the predicted bandwidth again far exceeds the actual due to

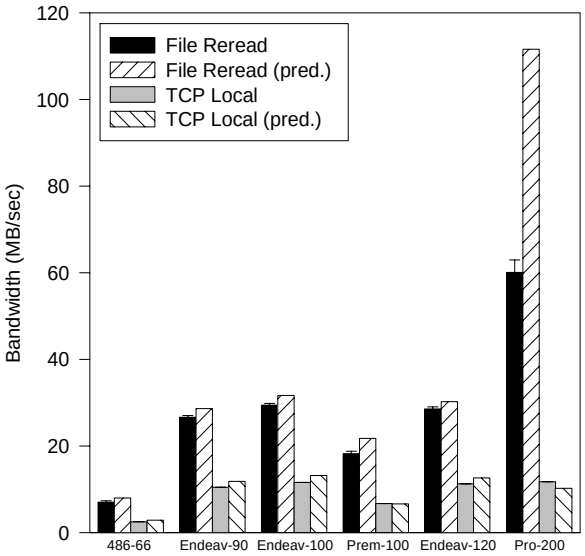

Figure 5: File Reread (64k buffers) and TCP Bandwidth (1MB buffers) Performance. File reread requires three memory references for each word of data read: a two-reference copy from the buffer cache to user space, and a final read as the user program touches the data transferred. The predicted file reread numbers were derived from this decomposition. The measured results fall short of the predictions due to cache contention and system-call overhead, and (for the *Pro-200*) extra bus negotiation cycles. The TCP benchmark performs three copies with buffers greater than the size of the cache, so in all cases we see about the predicted value, one-third times the bcopy bandwidth. The *Pro-200* performs better than predicted due to increased performance on the packet-checksumming component of the benchmark.

bus turnaround time that was not included in the raw read bandwidth, this machine still far outclasses the Endeavor-based Pentiums on this test despite its slower main memory system and poor bcopy performance. The reason for this is that the 64KB transfer buffers all lie entirely in the fast write-back L2 cache for the duration of the benchmark. If the read request size is increased to 1MB, larger than the 256KB L2 cache, the performance drops by a factor of two, as the buffers fall out of the L2 cache. The same effect can be seen in the TCP bandwidth test: using 64KB socket buffers instead of the default 1MB buffers increases performance 200% from the value in Figure 5. These results suggest that a fast write-back L2 cache can provide a significant advantage to an application that processes large amounts of data using a single buffer *that fits within the L2 cache*; if the buffer is large or if the application does not reuse the same buffer repeatedly, the overhead of faulting-in cache lines over a slow bus significantly reduces the write-back advantage.

Thus, in the case of the bulk data transfer primitives that an OS-dependent application might use, our decomposition is complete: the user-visible primitives of cached file reread and TCP data transfer are nearly entirely dependent on the memory system, and therefore it is features of the memory system that will most affect the performance of these primitives. The Endeavor-based Pentium results imply that for high-bandwidth applications, a main memory system based on fast DRAM technology (such as EDO memory) is essential. The *Pro-200*'s performance suggests again that eliminating unnecessary cache-coherency and bus transaction overhead will increase its performance greatly. It also suggests that intelligent, non-blocking, write-back caches are a net performance win both when reading large amounts of data and when handling data in units small enough to be cached, despite the delays that can be incurred in fetching lines upon write. In fact, analysis of these benchmarks

with the Pentium Pro hardware counters shows that, while transferring large amounts of data, the Pentium Pro rarely needs to read entire lines before writing into them, as the cache is intelligent enough to accumulate line-sized writes. Thus we conclude that large improvements to the CPU's execution unit (as in the *Pro-200*) may have a much less visible effect on high-bandwidth applications than small improvements in the memory subsystem (i.e., the use of a non-blocking write-back or pipeline-burst cache). Since multimedia applications and even the X Windows server transfer large quantities of data via the application primitives we have considered here, making these simple memory-system optimizations is crucial to attaining high performance.

4.2 Process Creation

With the bulk data transfer primitives as an example of how *hbench:OS* can perform a full performance decomposition from the bottom up, we now move on to consider the case of an OS primitive for which we can create a top-down decomposition with *hbench:OS*. The primitive that we will consider is process creation, because UNIX users and applications treat processes as the fundamental unit of work on the system; similarly, many server applications fork a new process for each request that they receive. Process creation consists of two components. A *fork* duplicates the currently running process and an *exec* overwrites the current process with the newly created process. Executables may be statically linked or dynamically linked; dynamically linked executables must resolve their library references at exec time. *hbench:OS* measures three methods of process invocation: a simple fork, a fork and exec, and process invocation via the shell. We run each of the two latter tests twice, measuring both static and dynamic linking of the target program ("hello-world").

In order to isolate each component of process creation, we decompose the more complex operations (e.g., process creation via /bin/sh) into the fundamental operations that we can measure. By subtracting fork latency from the combined fork and exec, we derive exec latency. The /bin/sh case is somewhat more complicated in that it consists of:

- fork current process,
- exec /bin/sh,
- fork /bin/sh, and
- exec hello.

If the shell and our target program were of comparable size, we would expect the /bin/sh case to be twice as slow as fork and exec. However, /bin/sh is significantly larger than our target program, so its fork and exec latencies are greater than those of the target program, causing the total /bin/sh latency to be somewhat greater than twice the combined simple fork and exec times.

We begin our analysis of these results with the one process creation metric that *hbench:OS* directly exposes: the cost of a fork, represented by the lowest sections of the bars in Figure 6. Comparing the fork cost across the suite of test machines reveals that the fork cost is primarily dependent on the memory system, although it does have a small clock-speed or CPU dependent component. Both the *486-33* and the *486-66* (which share the same memory system) demonstrate approximately the same fork latency; the *486-66* is slightly faster, highlighting the small CPU component. Similarly, the *Prem-100*, with its slower memory system, exhibits larger fork latency than its Endeavor-based counterpart. The Pentium Pro outperforms the Pentium due to both the CPU component of the test and the small-write-biased nature of the test: a fork on NetBSD/x86 involves building and zeroing a page table structure that fits in the Pentium Pro's write-back L2 cache.

Next we consider the exec latency, which we decomposed from the high-level *hbench:OS* tests by subtracting the fork latency from the fork+exec latency. The same comparisons as above reveal primarily a memory-system dependence for the static case: the OS must demand-copy the executable from the in-memory file system buffer cache. In this case, the CPU dependent component is minimal, and most likely results from the actual execution of the hello-world program. The exec latency in the dynamically-linked case has quite a different pattern. First, the latency is exceptionally large due to the cost of loading and mapping the shared libraries. We still observe a significant memory-system dependency, but the CPU dependent component has grown due to the need to build and initialize jump tables for the libraries. This is again evident in comparisons between the *486-33* and *486-66*, and between the *Prem-100* and the *Endeav-100*: in the first case, the *486-66* outperforms the *486-33*, but not as much as pure CPU scaling would suggest; in the second, the 100 MHz Pentium on the Endeavor motherboard outperforms the same chip on the Premiere motherboard. Again, since the benchmark fits in its L2 cache, the *Pro-200* performs well on this test, but still not significantly better than the Endeavor Pentiums.

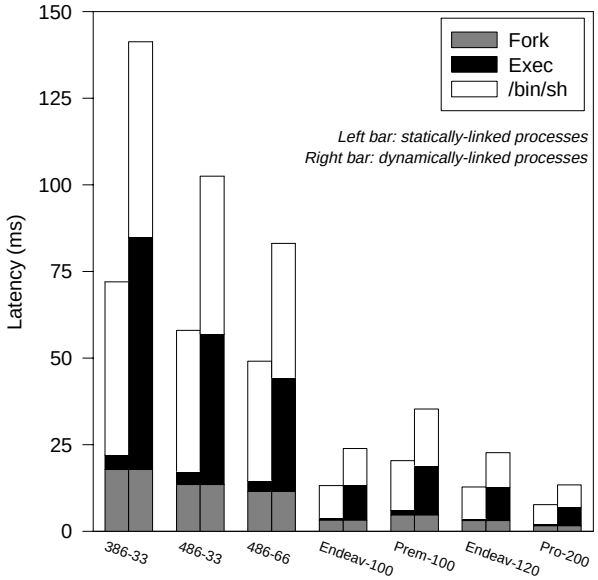

Figure 6: Process Creation Latencies. The total height of each bar represents the time to run "hello-world" via /bin/sh. The left bar in each group corresponds to the case where "hello-world" is statically linked; the right bar corresponds to a dynamically-linked "hello-world". The total process creation latency decomposes into three fundamental latencies: the latency of a simple fork(), the latency to exec() the hello-world process, and the latency introduced by the shell (which consists of the time to exec() /bin/sh and the latency of the fork() performed by /bin/sh). These are indicated by the subsections of each bar.

Finally, having used the high-level *hbench:OS* tests to extract the fork and exec latencies, we use these results to complete our decomposition by analyzing the overhead imposed by using the shell to invoke the hello-world process via the system() routine. If we consider the decomposition in Figure 6, we see that the /bin/sh overhead includes only the time involved in exec'ing /bin/sh and forking /bin/sh; the original fork to start the shell and the exec of hello-world are already accounted for. Comparing the /bin/sh overhead across the various test machines, we again see a heavy memory system dependency, just as we saw for the statically-linked fork and exec latencies. This is because the fork and exec components of the /bin/sh overhead are directly related to these fork and static-exec latencies, since under NetBSD, /bin/sh is statically linked. However, the magnitude of the /bin/sh overhead is significantly greater than the magnitude of the static hello-world fork and exec; this is because the shell binary is almost seven times larger than the statically-linked hello-world binary, so the memory component in-

volved in paging in the executable and initializing its mappings is proportionally larger. When "hello-world" is dynamically linked, the shell overhead is only slightly larger due to the extra overhead of managing the shared library mappings when starting the shell.

Thus, in process creation, we have an example of an alternate method of performance decomposition via *hbench:OS*: in this case, we began with the measured performance of high-level operations (process creations) and massaged these data to extract the performance of the primitive operations upon which the high-level operations are based (such as fork and exec latencies). We then applied our cross-platform comparison technique to understand the hardware basis for the performance of the low-level primitives. The inescapable conclusion is that, yet again, the memory system dominates performance: all of the primitive latencies, and the high-level process creation latencies, depend primarily on the memory system, and include only a small CPU-dependent component. Thus the Pentium Pro's performance margin over the Pentium systems is due not to its advanced out-of-order core, but rather to its speedy on-chip cache system.

4.3 Signal Handler Installation

Finally, we present an example case in which *hbench:OS* fails to offer the tools for any multilayer performance decomposition: this is the case of signal handler installation, again a frequently-used function in modern applications (the Apache web server executes this system call, on average, three times per accepted connection, according to our profiling results). We show how, in this case, our alternate methodology of cross-architecture comparison allows us to obtain useful results even where the *hbench:OS* tests are lacking. Figure 7 plots the results from this benchmark on some of our test machines.

The results indicate that, with the exception of the *Pro-200*, signal handler installation latency is entirely dependent upon CPU clock rate within each CPU class. The *Endeav-100* and *Prem-100* both perform almost the same number of installations per second despite their disparate memory systems; the *486-66* doubles the *486-33*'s performance, and the *Endeav-120* performs about 120/90 more installations per second than the *Endeav-90*. Comparisons between CPU classes suggest that there is a subtle performance dependence on more than the raw instruction execution rate: the 386 achieves less than half the performance of the 486 at the same clock speed, and the 486s obtain only about half the performance that a Pentium would at the same clock rate. This suggests that the performance of signal handler installation, in fact, depends on the L1 cache speed: the 386 has no L1 cache, so its performance is halved compared to the 486; the 486 requires two stall cycles to access data in its L1 cache compared to the Pentium's one cycle, accounting for the factor of two performance gain in the Pentium class. This hypothesis is confirmed by source code analysis, profiling, and analysis with the Pentium hardware counters: the signal handler installation system call spends the bulk of its time copying small, easily-cacheable data structures to and from user space. The only mystery that remains, then, is the *Pro-200* result, which is 27% worse than would be expected based on cycle time alone, and 42% worse than would be predicted based on the *Pro-200*'s L1 cache bandwidth. Without the underlying low-level tests that a full decomposition might offer, we have no way to understand this anomalous result, and can only speculate that for some reason, perhaps when the OS switches from user to kernel mode, some internal CPU state (such as the branch target buffers) is being flushed, or else the CPU is incurring more unnecessary cache-coherency overhead. Even the Pentium Pro hardware performance-monitoring counters do not shed light on this bizarre result.

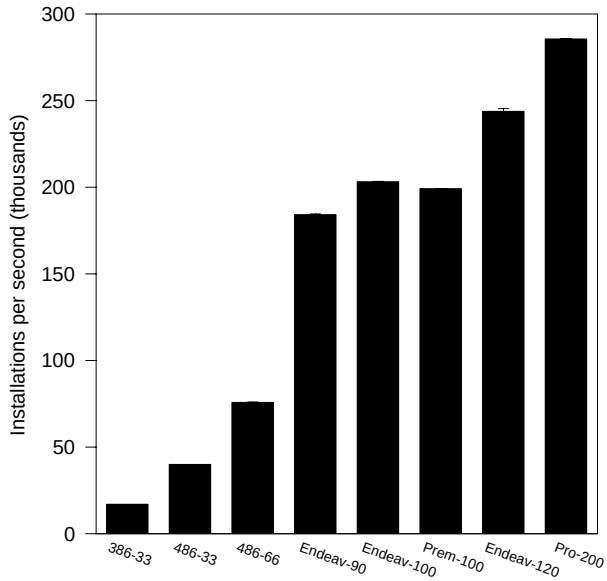

Figure 7: Signal Handler Installation. This graph plots the number of signal handler installations that each of our test machines can perform in one second. The results seem to scale with the CPU clock rate, but the factor of two performance difference between the *386-33* and the *486-33*, and a similar jump from the 486's to the Pentiums, suggests that the results of this test are determined by L1 cache performance.

Thus, in the case of signal handler installation, we see that we can obtain generally useful results even when *hbench:OS* does not include the capability to decompose the performance of the interesting high-level functionality into lower-level primitives. Here, we can conclude that lower-latency L1 caches are the key to improving signal handler installation performance. However, we are left at a loss when anomalous results (such as the *Pro-200*'s) appear, since we have no lower-level tests to use as a basis for understanding the unexpected results.

5 Conclusions

With the modifications that we made to support increased parameterization and statistical rigor, *hbench:OS* succeeds in most cases as a tool for detailed analysis and decomposition of the performance of operating system primitives. In our example cases of bulk data transfer and process creation latency, *hbench:OS* provided enough detailed tests to build a hierarchical decomposition of performance from the complex, application-visible primitives at the top to the hardware and architecture at the bottom. Where *hbench:OS* failed to provide the tools for such a decomposition, as in the case of signal handler installation, our alternate methodology of cross-machine comparison was able to uncover the general architectural features upon which the OS primitive in question depended. However, in these cases the tools provided by *hbench:OS* are still inadequate: to fully understand anomalous results requires more information than the high-level benchmarks can provide. We feel that it will be possible to apply profiling techniques to such high-level primitives to determine their performance decomposition; from these profiling results a benchmark could then be constructed to isolate, and characterize in terms of hardware performance, each of the individual dependencies.

When we applied our *hbench:OS*-based analysis techniques to NetBSD, we were surprised to find that, for nearly all high-level OS primitives upon which modern applications depend, it is the memory system, and especially the access path to the off-chip memory system, that dominates performance. Particularly intriguing were the results from the *Pro-200*, the Pentium Pro-based machine. Despite major improvements to the processor's execution pipeline and

cache subsystem compared to the Pentium, the Pentium Pro did not significantly outperform the Endeavor-based Pentiums on many of the tests. In fact, the addition of multiprocessor coherency support and transaction-based bus protocols into the CPU, and the resulting poor external memory system bandwidth, seem to have essentially negated any performance advantage that the CPU's advanced execution core provides. Essentially, Intel's multiprocessor optimizations have crippled the performance in single-CPU systems.

Also illuminating was the comparison between the Endeavor- and Premiere-based Pentiums; the Endeavor, with pipeline-burst cache and EDO support, outperformed the Premiere system by nearly a factor of two in many cases, MHz for MHz. While researchers have known for several years that a high-performance memory subsystem is important to OS performance [1][12][13], it seems that, at least for the x86 architecture, the industry's focus on the processor's pipeline and cache subsystem has been misdirected. For example, Intel's high-end Pentium server motherboard, the *Xpress*, eschews the advantages of EDO or synchronous DRAM for a large 1MB L2 cache since the larger cache produces higher SPECmark ratings [7]; our results indicate that to improve the performance of OS-dependent server applications, Intel would have done better by engineering a higher-performance EDO DRAM-based system than by focusing on the caches.

High-bandwidth OS-dependent applications have become common—witness the explosion in the number of web servers running today. It is time for vendors to turn their focus from SPECmarks and application-level performance to OS performance. Tools such as the *hbench:OS* suite offer a solid foundation for evaluating OS performance, and can bring to light surprising facts about the design of today's systems, as we have seen in our analysis of NetBSD on the x86 platform. It is tools such as these that will provide the understanding of the architectural dependence of OS performance necessary to build tomorrow's high-performance hardware.

6 References

[1] Anderson, T., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., Wang, R., "Serverless Network File Systems," *Proceedings of the Fifteenth Symposium on Operating System Principles*, Copper Mountain, CO, December 1995, 109–126.

[2] Brown, A., "A Decompositional Approach to Performance Evaluation," Technical Report TR-03-97, Center for Research in Computing Technology, Harvard University, 1997.

[3] Chen, B., Bershad, B., "The Impact of Operating System Structure on Memory System Performance," *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, NC, December 1993, 120–133.

[4] Chen, B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., Smith. M., "The Measured Performance of Personal Computer Operating Systems," *Proceedings of the Fifteenth Symposium on Operating System Principles*, Copper Mountain, CO, December 1995, 299–313.

[5] Computer Systems Research Group, University of California, Berkeley, "4.4BSD-Lite CD-ROM Companion," O'Reilly and Associates, 1st Edition, June 1994.

[6] Gloy, N., Young, C., Chen, J., Smith, M., "An Analysis of Dynamic Branch Prediction Schemes on System Workloads," *Proceedings of the International Symposium on Computer Architecture*, May 1996.

[7] Gwennap, L., "Pentium PC Performance Varies Widely," *Microprocessor Report*, 2 October 1995, 14–15.

[8] Intel Corporation, *Pentium Pro Family Developer's Manual, Volume 1: Specifications*, Order number 242690-001, Mt. Prospect, IL, 1996.

[9] Intel Corporation, *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*, Order number 242692-001, Mt. Prospect, IL, 1996, Appendix B.

[10] Maynard, A., Donnelly, C., Olszewski, B., "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads," *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994, 145–157.

[11] McVoy, L., Staelin, C., "lmbench: Portable Tools for Performance Analysis," *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, January 1996, 279–295.

[12] Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" *Proceedings of the 1990 Summer USENIX Technical Conference*, Anaheim, CA, June 1990, 247–256.

[13] Rosenblum, M., Bugnion, E., Herrod, S., Witchel, E., Gupta, A., "The Impact of Architectural Trends on Operating System Performance," *Proceedings of the Fifteenth Symposium on Operating System Principles*, Copper Mountain, CO, December 1995, 285–298.