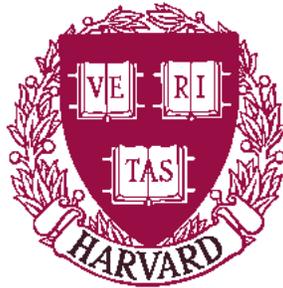**A Decompositional Approach to
Computer System Performance Evaluation**

Aaron B. Brown

TR-03-97

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

A Decompositional Approach to Computer System Performance Evaluation

A Thesis presented

by

Aaron Baeten Brown

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 7, 1997

Table of Contents

Chapter 1

Introduction

Throughout the history of computers, a primary design goal in all systems has been to achieve maximum performance. When digital computing was in its infancy, performance was measured in terms of throughput of scientific calculations: in the mid-1940s, the then-new ENIAC's most-touted design feature was its ability to calculate artillery trajectory tables over 1,000 times faster than earlier electromechanical computers [18]. Surprisingly, this traditional approach to performance evaluation and optimization remains in widespread use today. Admittedly, one does not see today's processor designers optimizing their commodity CPUs for use in anti-aircraft guns, but it is still true that the performance evaluation suites in widespread use are based either on models of scientific computation or on a systems programming or hardware design workload [23][24]. These models are becoming progressively less valid for today's applications: modern users are focusing more and more of their attentions on applications heavy with multimedia and networking content. One need not look farther than the explosive success of the world-wide web and the plethora of multimedia extensions and protocols that it has spawned to see this trend in action.

The demands that such modern applications make on the hardware and software subsystems that support them are very different than those simulated and evaluated by the traditional set of performance analysis tools. 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 (OS) kernel, an area of the system almost completely ignored by such traditional performance-evaluation tools. As an illustrative example, consider what must undoubtedly be today's leading server application: the web server. Web servers have been shown to spend over 85% of their CPU cycles running operating system code [4]; in contrast, the near-ubiquitous SPEC benchmarks execute less than 9% of their instructions in the OS kernel [7]. The figures in Table 1 show that the web server is not unique: the other typical multimedia, commercial, and GUI workloads listed similarly spend between 20% and 90% of their instructions in the ker-

	Benchmark	User	Kernel	%-Kernel
		Dynamic Instruction Counts (in millions)		
SPEC92	compress	70.9	3.5	4.7
	espresso	332.3	6.2	1.8
	gcc	145.7	4.3	2.9
	022.li	121.4	4.6	3.7
	072.sc	81.8	7.8	8.7
	089.su2cor	385.2	2.2	0.1
IBS	jpeg_play	138.9	12.2	8.1
	mpeg_play	77.8	21.6	21.7
	verilog	36.4	10.6	22.6
	video_play	15.8	36.7	69.9
Maynard	TPC-A	not available	not available	40
	TPC-C	not available	not available	43
	Netperf	not available	not available	97
	Laddis	not available	not available	100
	Kenbus	not available	not available	23
	Sdet	not available	not available	50
Chen	Ghostscript	538.7	49.3	8.4
	Web	163.6	764.1	82.4
	Wish	193.6	23.0	10.6
		By Cycle Count (in millions)		
Chen	Ghostscript	1275.9	242.2	16.0
	Web	568.5	3793.4	87.0
	Wish	492.5	138.5	21.9

Table 1: Percent Time Spent in Kernel Code. The SPEC benchmarks spend significantly less time in the kernel than a large number of commercial and multimedia workloads, represented here by the IBS, Maynard, and Chen suites. The SPEC and IBS measurements are from [7], the Maynard workloads are from [12], and the Chen measurements are from the data that accompany [4].

nel. Amdahl's law tells us that if we want modern applications such as these to run quickly, the operating system must run quickly as well. Since traditional performance models essentially ignore the operating system and modern OS-dependent applications, a need has arisen for new tools and methodologies that direct their attention at the performance of the OS kernel and the applications that depend on it. The goal of this thesis is to fulfill that need.

In order to develop new tools and methodologies for OS measurement, we must first recognize the unique role played by the OS in a modern system, and consider how existing performance analysis techniques fail to take this role into account. These understandings will provide the foundation for a new set of tools and methodologies that accurately characterize the true nature of OS and OS-dependent application performance.

The modern operating system is, above all else, a provider of abstraction: it takes the raw resources of the low-level hardware and abstracts them into a well-behaved interface for use by the system's resource consumers: user applications and the users themselves. This is a very different role than is played by any other part of the computer system, and, as a result, the designer of an OS kernel faces a unique set of performance challenges that are not conducive to traditional performance analysis techniques. One such challenge arises from the inherent performance penalty imposed by abstraction: the OS designer needs to build abstractions that properly virtualize resources yet at the same time preserve as much of the hardware's raw performance for the end user. A more significant challenge comes in analyzing the abstractions selected for and built into an operating system, for operating system performance cannot be studied in isolation. Because the OS is primarily an abstraction provider, its performance depends directly upon the capabilities of the low-level hardware interface that it is abstracting, as well as on the way that the abstractions it creates are used by applications (since not all applications exercise the same OS abstractions). Thus any methodology for analyzing OS performance must take into account both the OS's dependence on features of the underlying hardware architecture as well as on the patterns of application load that indicate which abstractions are important.

Traditional methods for performance analysis are poorly suited to this task. They break down into three basic categories: macrobenchmarking, kernel profiling, and microbenchmarking. Macrobenchmarking, or timing runs of real applications, can be useful for

measuring end-to-end performance on a specific workload, but does not produce enough detail in its results to unravel the complex interactions between the application load and the OS performance, or between the OS and the hardware. Kernel profiling, in which the amount of time spent in each kernel function is measured, gives a highly detailed picture of the kernel's execution, but does not provide the time-sequence information necessary to correlate an application's varying use of kernel abstractions with potential kernel perfor- mance bottlenecks. Additionally, standard kernel profiling[1] cannot accurately associate I/O and sleep time with user-process requests, and thus does not provide an accurate mechanism for associating user-visible latency with time spent in kernel routines. Finally, we are left with microbenchmarking, in which the performance of very small pieces of the OS kernel's abstraction layer are measured individually. By itself, microbenchmarking only measures the efficiency of individual kernel abstractions; it ignores how the abstrac- tions are used by user applications, and thus, like the other two techniques, does not indi- cate how to evaluate and optimize the performance of kernel abstractions under realistic application load.

With an understanding of both the demands of OS performance measurement and the flaws in existing measurement techniques, we can now return to the problem of devel- oping tools and methodologies appropriate for analyzing OS performance in its true con- text, intertwined with hardware performance and OS-dependent application load. The approach that we propose for accomplishing this goal is grounded heavily in the notion of an *abstraction hierarchy*. We can view a complete computer system as a continuum of abstractions stretching from the silicon of the hardware up through the operating system and ending at the application's user interface. The abstractions within this continuum form a natural hierarchy that complements traditional modular design methodology: the hardware provides abstractions of computation and I/O upon which the OS is built; the OS in turn provides virtualized abstractions of physical resources upon which applications are built; finally, applications provide high-level abstractions that organize the OS's abstractions and with which users interact. Figure 1 presents this idea graphically.

---

1. Some of the problems with traditional kernel profiling are remedied in *pkprof,* a kernel profiler that associates all acquired performance data with user-process requests [16]. However, *pkprof* is currently available only on stock 4.4BSD-Lite running on the MIPS-based DECstation platform, and thus does not provide a general solution to the kernel-profiling problem.
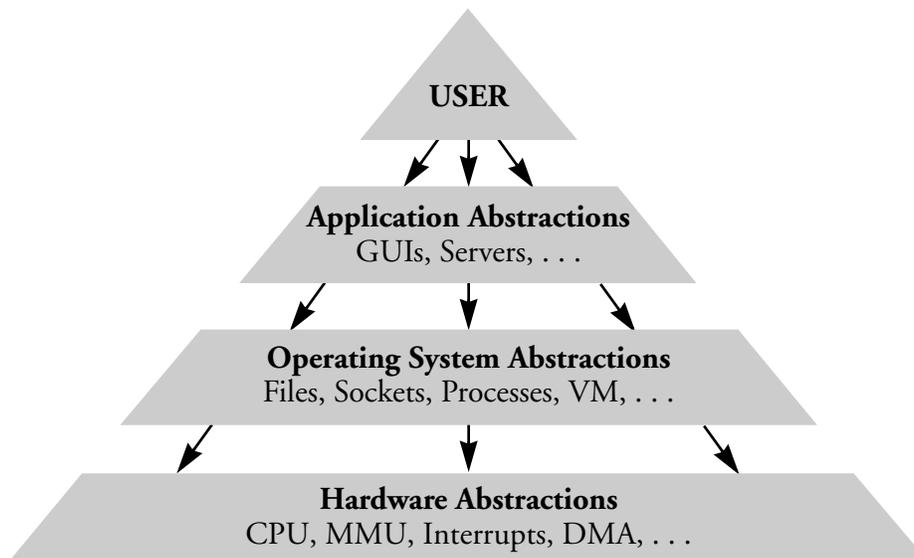
Figure 1: Computer System Abstraction Hierarchy. The gray trapezoids represent layers of abstraction; the arrows represent performance dependencies between layers. The user interacts with the application's abstractions, which in turn depend on the operating system's abstractions, which then depend on the hardware's abstractions. This same abstraction hierarchy transforms easily into a performance hierarchy.

The structure of a system's abstraction hierarchy is inherently intertwined with the system's performance, for the performance of any given abstraction depends on the performance of the abstractions upon which it is layered. Thus it is easy to transform the abstraction hierarchy into a *performance hierarchy,* in which the nodes have the same structure as those in the abstraction hierarchy, but the connections between nodes represent the performance dependencies between various abstractions: at each abstraction boundary, the connections to higher level abstractions represent the performance demands of the part of the hierarchy above the boundary, while the connections to lower level abstractions represent the performance resources of the portion of the hierarchy below the boundary.

The performance hierarchy provides an ideal framework for understanding operating system performance, for it captures in one construct the details of the performance interactions between the hardware, operating system, and application. As the application runs, it exercises paths that weave their way from the apex of the hierarchy (where the application's highest-level abstractions or operations are found) through the OS's abstractions and down to the base of the hierarchy (which consists of the fundamental hardware abstractions). By looking at how these paths intersect the hierarchy at the level of the OS-exported abstractions, the operating system designer can determine which abstractions are

critical for the application's performance, and can focus optimization effort appropriately; the application designer can use the same information to optimize his or her application's use of potentially-slow operating system abstractions.[2] Analogously, by looking at how the paths cluster at the bottom of the performance hierarchy, the hardware designer can see where to tune the hardware to maximize the operating system's (and therefore the application's) performance; the operating system designer can use the same data to determine where optimizations to the kernel are warranted.

Thus the first task in understanding the performance of a specific operating system is to construct the performance hierarchy that includes the hardware on which it will run and the primary applications that it will support. The primary technique that we have used to do this is *performance decomposition*, in which the performance of each abstraction in a given level of the performance hierarchy is decomposed into the performance of abstractions in the next-lower level of the hierarchy. This is usually accomplished in one of two ways. Where the performance of an abstraction can be derived analytically from the performance of those below it, the decomposition can be taken directly from the analytic model. In the more common case where analytic models are not available, the dependencies must be reconstructed by varying the performance of the lower-level abstractions (usually by manipulating the hardware platform) while observing the response in the performance of the higher-level abstraction. Chapters 2 and 3 introduce tools and methodologies that help perform such performance decompositions in the operating system and application domains, respectively; they also present case studies that demonstrate the use of the techniques and some interesting sample results.

The technique of developing a performance hierarchy by performance decomposition is a solution to only part of the problem of operating systems measurement, however. While useful to the system designer in that it provides enough information to show where optimizations need be applied, the performance hierarchy approach produces too much detail to be of much use to the less-sophisticated consumer. These consumers include anyone else who cares about operating system performance, from application designers to system administrators; one might think that the kinds of performance results desired by such

---

2. In the context of an extensible operating system, such data would provide important clues as to where specialized extensions and interfaces should be installed.

consumers are mutually incompatible with the details that the performance hierarchy produces for the system engineers. If we consider the types of information that would be useful for such consumers, however, a pattern quickly emerges: all the measurements of interest can be taken at the major abstraction boundaries in the performance hierarchy. For example, an application designer might want to know the performance characteristics of the major abstractions exported by a particular operating system/hardware pairing; these are obtained by taking a horizontal cross-section of the performance hierarchy at the lower (OS) side of the boundary between operating system and application. Similarly, a hardware designer interested in optimizing for the operating system would most likely be interested in a cross-section of the hierarchy at the upper side of the OS-hardware boundary. Finally, a system administrator interested in picking the best OS/hardware pairing to support a given application (such as a web server) might find two cross-sections useful: one at the upper side of the OS-application boundary to characterize the application's demands on the operating system (independent of the OS/hardware platform) and one at the lower side of the same boundary (to characterize the operating system's ability to satisfy the application's demands).

Thus, to complete our methodology of OS performance analysis, we must find a way to distill the detail of the performance hierarchy into a characterization of performance interactions at a given abstraction boundary; ideally, this characterization should take the form of a vector of performance demands by the part of the system above the boundary, a vector of performance resources offered by the part of the system below the boundary, and a way to combine these vectors to get a single, simple metric representing how well the demands are satisfied by the available resources. Although a complete solution to this problem at all abstraction boundaries is well beyond the scope of this thesis, in Chapter 4 we consider the specific case of the application-OS boundary, and derive a methodological approach that transforms the performance hierarchies for an application and an OS into a single metric that predicts user-visible system performance.

Thus throughout this thesis we present methodologies and tools explicitly designed for building and analyzing the performance hierarchies that define modern system performance. We begin in Chapter 2 with techniques to characterize and link together the foundation layers of the hierarchy: the hardware and the operating system. Next, in Chapter 3,

we extend the hierarchy to include the performance abstractions of OS-dependent user applications by presenting techniques for decomposing user-visible application performance into dependencies upon the operating system. Finally, in Chapter 4 we take the performance hierarchy obtained by the techniques in the earlier chapters and discuss a methodology that is capable of reducing the intricate detail of the performance hierarchy into a set of simple characterization vectors that can be transformed into a single application-specific metric for gauging or predicting operating system and application performance. The result of the analysis in these three chapters is thus a methodology that restores relevancy and rigor to operating system measurement: it discards *ad hoc*, artificial measurement in favor of a well-defined blueprint for analyzing operating system performance in the context of real hardware and real application load, and, as a result, produces metrics that can be used in a meaningful way for system performance evaluation and comparison.

Chapter 2

Decomposing the Performance of the Operating System Kernel

The first step in understanding operating system performance is to understand how the performance of the operating system's major abstractions depends on the performance of the underlying hardware platform. Gaining this understanding is a prerequisite for constructing the lower half of the system performance hierarchy, which encompasses both the hardware and operating system layers. The methodology that we have chosen to accomplish this task is based heavily on the notion of performance decomposition, and relies on microbenchmarking for its tools. Although we have seen how microbenchmarks used in a vacuum do not provide the information needed to understand the interactions and performance characteristics of a given system, when combined with the performance decomposition technique they provide the means that we need to achieve the desired characterization of the lower half of the system performance hierarchy.

The microbenchmarks that we use (*hbench-OS,* described in Section 2.2) divide roughly into two areas: one set quantifies hardware capabilities and performance (e.g., memory bandwidth), while the other measures the primitive functionality that is exported from the kernel to applications (such as the system calls, process creation, and file/network access). The results produced by these two sets correspond naturally to the bottom layers of the system performance hierarchy; Figure 2 depicts the structure of this piece of
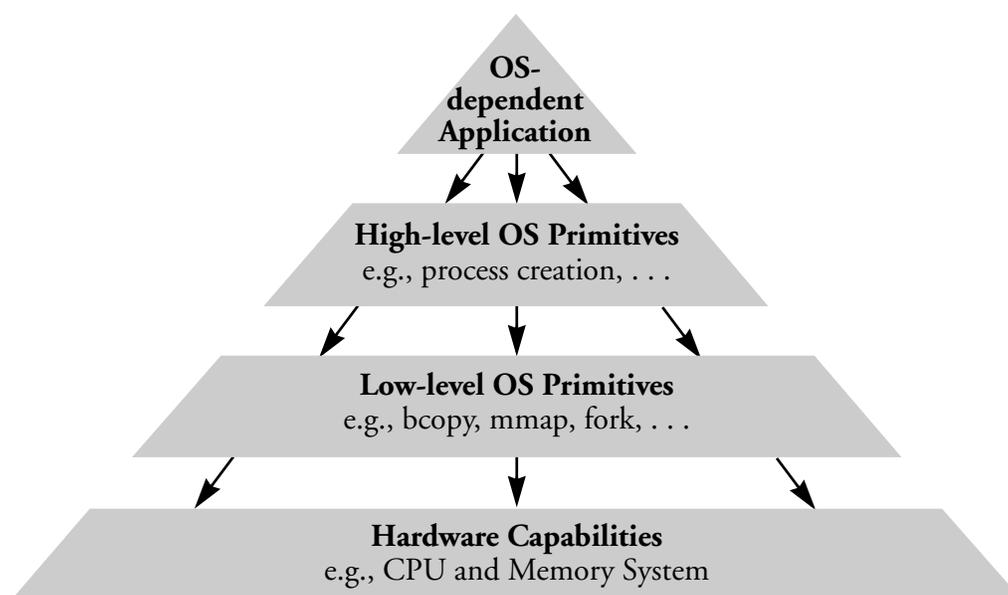
Figure 2: Decomposition of OS Performance via *hbench-OS* Primitives. The performance of OS-dependent applications can be decomposed into high-level OS-provided services and primitives, which can in turn be decomposed into low-level kernel primitives (that may themselves be used by applications). These low-level kernel primitives can, in turn, be decomposed again into hardware capabilities. In many cases, *hbench-OS*'s suite of tests allows us to measure and relate the lower three levels of this hierarchy.

the hierarchy as a pyramid of relationships between layers representing components of OS performance.

The performance decomposition methodology enters when we need to reconstruct the interdependencies in the OS/hardware portion of the performance hierarchy, a task for which the microbenchmarks themselves do not provide any guidance. As described in Chapter 1, this task can be approached from two directions, analytically or experimentally. We will use both techniques, but will direct our focus on the experimental model, for it is rare that the performance of such a complex system as the combination of OS and hardware can be predicted analytically. The key idea in performing a performance decomposition experimentally is to use controlled variation of the performance of a low-level primitive or abstraction in order to gauge its impact on, and thus its connections to, higher-level primitives or abstractions. Thus our decomposition methodology consists of two steps: first, using our *hbench-OS* benchmarks to measure performance at each of the lower levels of the hierarchy while varying features of the hardware in a controlled manner (i.e., changing only one component of the test system at a time); 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 perfor-

mance of primitives in layers above and below it. Once the interaction between each pair of adjacent layers is understood, the hierarchy of performance dependencies can be reconstructed.

In many cases, our *hbench-OS* benchmark tests provide enough detail about the internal structure of individual layers of the hierarchy so that it is possible to analytically reconstruct the performance dependency paths that connect application-visible OS abstractions and primitives to hardware performance; this is especially true when the primitives being evaluated are related to bulk data transfer, as will be seen in the case study below. In other cases, however, the tests by themselves may only be capable of measuring the performance of the top-layer abstraction, and thus we are forced to rely on the experimental technique, bypassing the middle layers of the hierarchy to directly determine the hardware dependencies of the upper-layer primitives; as an example, we found it necessary to resort to this technique to relate the lowest-level primitives measured by *hbench-OS* (such as raw memory bandwidths) to features of the hardware architecture.

With this methodology for building the lower levels of the system performance hierarchy in hand, we are now ready apply it to a real system. Before doing so, however, we first present some other approaches to OS performance evaluation, and discuss how our decompositional methodology differs from these traditional techniques. Next, in Section 2.2, we describe in greater detail the *hbench-OS* tools that we have developed for measuring the performance of OS primitives. These tools are based on the *lmbench* benchmark suite [14], but have been significantly enhanced and modified; we describe these modifications as well. Finally, in Section 2.3, we apply our tools and methodology to the task of understanding and decomposing the performance of the NetBSD operating system [15] on the Intel x86 platform. In this last section, we also demonstrate the types of detailed conclusions that the performance-hierarchy approach allows us to draw about the interactions between OS and hardware performance, and how these conclusions provide guidance for future hardware and software tuning; we defer discussion concerning the OS's impact on application performance to Chapters 3 and 4.

2.1 Related Work: Benchmarking Operating Systems

The operating systems research community has not ignored OS performance; on the con-

trary, there is a large body of work aimed at understanding OS performance and its hardware dependencies. The typical approach that has been taken is the microbenchmarking approach: various OS primitives are measured across a wide range of hardware platforms, and any trends in the data are used to draw conclusions relating OS performance to hardware performance. Probably the most frequently cited example of this type of work is Ousterhout's 1990 paper "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" [17] in which Ousterhout uses a set of tests (originally designed to measure the performance of the Sprite operating system) to analyze the performance of OS primitives across a range of then-common processor and system architectures, primarily to determine the performance impact of the move from CISC to RISC architecture. 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 capabilities of the underlying hardware and to use that characterization to understand the performance of higher-level kernel primitives; thus they are not useful for the performance decomposition approach.

Anderson et al. pursued a similar approach in order to examine the interplay of hardware architecture and the operating system in a multiprocessing microkernel design [1]: they used a set of *ad hoc* microbenchmarks to perform a cross-platform comparison of OS primitives targeted to microkernel bottlenecks (IPC and communications latency, thread overhead, etc.). Again, the benchmarks they used were not complete enough to provide the detail necessary for constructing a full system performance hierarchy.

In 1995, McVoy improved the state-of-the-art in OS microbenchmarking 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) [14]. Although *lmbench*'s detailed tests offered the breadth and detail needed to decompose operating system primitives into their hardware-dependent components, McVoy used them primarily to draw conclusions about the relative merit of various operating system and hardware architectures. Additionally, the *lmbench* tests suffer from several statistical and methodological flaws that make them difficult to use as the basis of a performance

decomposition methodology; we discuss these flaws and our attempts to fix them in Section 2.2.

Although the microbenchmarking approach used by these researchers is similar to the one that we have chosen, there are two important differences. First, and most important, in this chapter we are not trying to make general claims about operating system performance. This type of conclusion can only be made when the OS is considered in the context of realistic application load, which we do later in Chapters 3 and 4; here, we are instead trying to reconstruct the dependencies of OS abstractions on hardware, without making any claims about the relative or absolute importance of any given abstraction. The other difference is in the methodology that we use to determine the performance dependencies: instead of trying to draw conclusions from comparisons across widely divergent hardware platforms, as do McVoy, Ousterhout, and Anderson, we rely on the principle of controlled experimentation, making the smallest possible changes to the hardware platform that produce noticeable performance differences. This technique is essential in order to control as many extraneous variables as possible; otherwise, it is impossible to isolate the effect of specific hardware features on OS performance, and the details of the connections between OS performance and hardware are lost.

2.2 Microbenchmark Tools: Revising *lmbench* into *hbench-OS*

Since microbenchmarks play such a key role in the decomposition-based methodology that we have developed, our first task in implementing the methodology was to construct a set of rigorous microbenchmark programs capable of producing accurate, reproducible results. We initially chose McVoy's *lmbench* [14], described above in Section 2.1, for the breadth of its tests and the detail they claimed to provide. However, as we began to use *lmbench* to study NetBSD's performance for the case study in Section 2.3, we found that it had several shortcomings as a tool for the detailed scientific study of OS-hardware interaction, which is what we needed to construct the lower levels of the system performance hierarchy. 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

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.
Null system call latency	Measures the latency of writing one byte to `/dev/null`, to approximate the cost of entering the kernel through the system call dispatcher.

Table 2: Summary of a subset of the original *lmbench* benchmarks [9]. Both the *lmbench* and *hbench-OS* suites include many other benchmark tests; those listed above are the most useful, and are the ones used in this thesis.

tools capable 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.

The specific problems that we encountered with *lmbench* ranged from minor (we found the output format of the benchmarks difficult to analyze) to substantial (with a reasonable compiler, the test designed to read and touch data from the file system buffer cache never actually accessed 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 thesis are summarized in Table 2; we refer the reader to McVoy's original *lmbench* paper [9] for a more detailed description of the benchmark tests discussed.

For the remainder of this section, we will 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.[3] Note that the Pentium Pro integrates the L2 cache into the same package as the CPU, while earlier CPUs use an external L2 cache.

2.2.1 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 (a software-simulated network interface that reroutes outgoing data back into the system's input queues, bypassing the real network interface and physical medium entirely). 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

---

3. A *cache* is a piece of small, high-speed memory that is used to reduce the access time to frequently used data normally stored in slow main memory (DRAM). As data is retrieved from DRAM, it is cached in the CPU's caches under the assumption that it will be reused in the near future; subsequent accesses to the data incur only the cache's low latency, not the DRAM's high latency. A system usually has two or more levels of caches; the L1 cache (closest to the CPU) is the smallest and fastest, and those at higher levels are slower but larger.

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.

2.2.2 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 $(100-2n)$%. 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 chapter, 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%.

2.2.3 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.

2.2.4 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.[4] 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

---

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

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 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%.

2.2.5 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 difficult 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.

2.2.6 New Output Format

Lmbench placed all of its output into one large file whose name identified the machine being benchmarked. In this format, the raw data was not very easy to extract for graphing or statistical analysis. Thus we decided to revamp the output format to make it easier to generate and process the raw data automatically.

In our new scheme, the user can specify in a configuration file exactly which tests are to be run; the driver script is automatically generated from this script. Each machine tested is assigned its own directory. Although the benchmark results are still stored initially in one large file, a script is provided to separate the large file such that the results from each parameterization of each benchmark are placed in a separate file whose name includes the name of the test and the parameters. Each file contains n numbers where n is the number of runs that were made of the benchmark. For example, if 10 runs were made of the context switch latency benchmark with 32 processes of size 16 KB, there would be 10 numbers in the file **lat_ctx_16k_32**. Several other files are used to hold the version/RCS and system configuration information. Each of the benchmark result files is processed by a script that sorts the values, strips the low and high tails, and calculates the average and standard deviation as described in Section 2.2.2; these processed results are placed back into individual files, which are then combined into a summary file. Another script produces an *lmbench*-style summary report from this. Thus the summary output is still available to the end user, but the raw data is preserved and made available to the researcher who wishes to use it; the use of the file-system namespace as a database of results allows easy selection of datasets for graphing or analysis of the raw data.

2.3 Case Study: Constructing a Performance Decomposition for NetBSD on the Intel x86 Platform

With both the decomposition methodology discussed at the beginning of this chapter and the *hbench-OS* tools described above in hand, we now turn to a case study in order to illustrate both the interaction between the tools and the methodology as well as the type of results that emerge in the process of constructing the lower levels of the system performance hierarchy. For our subject operating system, we chose NetBSD 1.1 [15], a derivative of the CSRG 4.4BSD-Lite release [5], which shares a common ancestry with many of today's commercial UNIX implementations. We selected NetBSD for its openness and its

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 writeback and on-chip	60 ns EDO 66 MHz	Pentium Pro (i440FX chipset)

Table 3: 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.

multiplatform support: having the source code meant that we could use kernel profiling and source code analysis to verify our techniques, and its multi-platform support provided the possibility of future cross-architecture comparisons. For our hardware test platforms, we selected eight machines from the Intel x86 architectural family: a 386, two 486's, four Pentiums, and one Pentium Pro. The hardware details of these machines are given in Table 3. 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. It thus allows us to build a set of performance hierarchies that include most of the hardware features that are typically found in modern microprocessors. 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.[5]

As described earlier in this chapter, the experimental approach to performance decomposition requires controlled variation of hardware features in order to isolate the dependencies of OS primitives on those features. The set of test platforms that we used were carefully selected to allow for such controlled variation. For example, the use of several different motherboards with the same CPU and several different CPUs with the same motherboard allowed certain comparisons to 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 specific comparisons that we used and the connections that they allowed us to form in the performance hierarchy are detailed in the following sections.

2.3.1 Bulk Data Transfer

We begin our study with an example of the best-case performance decomposition methodology, where the performance of high-level OS primitives can be tracked analytically all the way down to hardware dependencies and then verified experimentally. The most illuminating example of this is the case of bulk data transfer. We choose bulk data transfer as an illustrative OS primitive 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 55% 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 spaces. Since each of these data-access methods

---

5. 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).
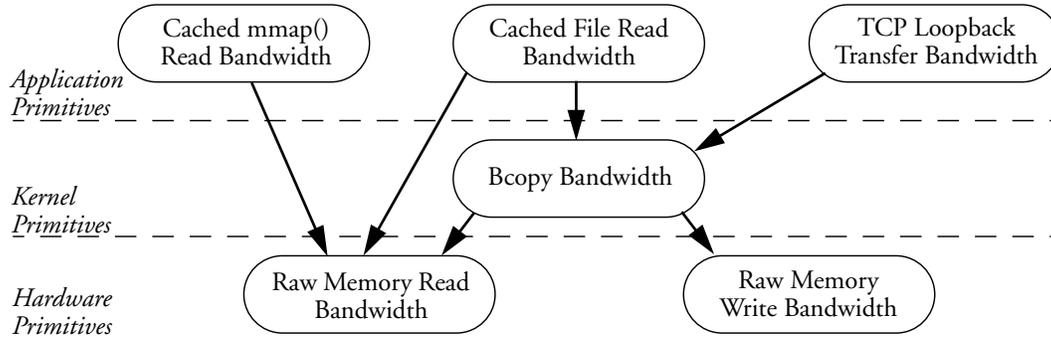
Figure 3: Decomposition of Application Data-access Primitives. All of the application-level primitives for bulk data access depend on the hardware's memory read bandwidth since they all touch data. File reading interposes the extra overhead of a cross-protection-domain bcopy to move the requested data 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; mmap adds virtual memory system overhead as it enters new mappings into the process's address space.

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 software loopback interface, which bypasses the physical network medium), we arrive at the decomposition shown in Figure 3. 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 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 that measure the bottom layer of Figure 3, 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
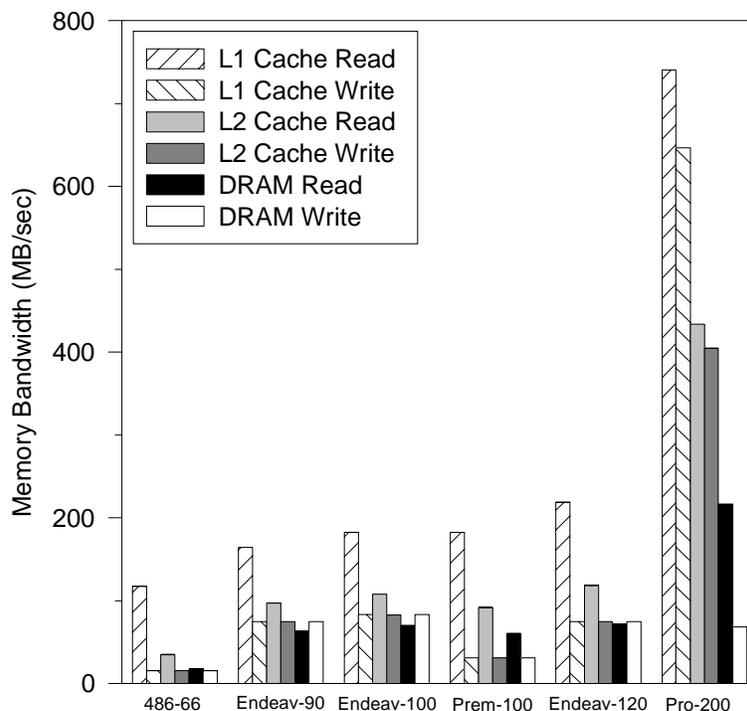
Figure 4: Raw Memory Bandwidth. The 64-bit, burst-capable memory bus of the Pentiums produces a factor of four improvement in L2 and DRAM read memory bandwidth from the i486 architecture. The combination of pipeline-burst cache and EDO DRAM gives the *Endeav-100* a significant performance advantage over 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.

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 4 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 4, 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 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; our later application measurements in Chapter 4 confirm this hunch.

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 [9]. 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 [11]. 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, mmap'd 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 5, we present the results of the non-cached bcopy test along with the half-harmonic means calculated from the raw bandwidth results in Figure 4; 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 DRAM write performance of the Pentium Pro memory system 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 pre-
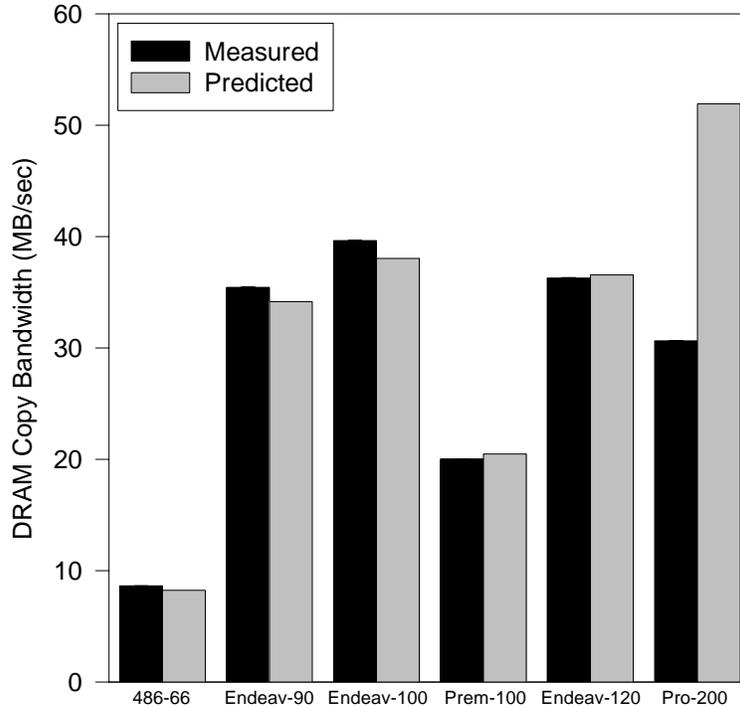
Figure 5: bcopy Bandwidth (2MB buffers). The memory systems determine performance on this benchmark: the predicted bcopy results (one-half of the harmonic mean of the read and write memory bandwidths) track closely with the measured numbers. When the 100 MHz Pentium is scaled to account for its higher 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 poor bcopy performance; the actual performance falls far short of the predicted performance because of bus turnaround time not accounted for in the read bandwidth.

diction, 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 FastStrings 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. From the decomposition presented earlier in Figure 3, 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
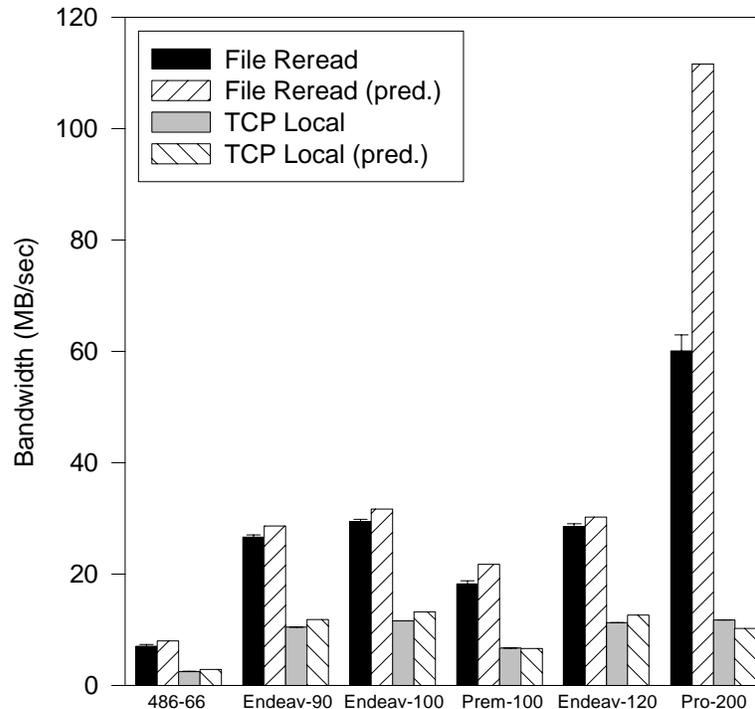
Figure 6: File Reread (64k buffers) and TCP Bandwidth (1MB buffers) Performance. File reread requires three memory references for each word of data read: a two-reference copy from the buffer cache to user space, and a final read as the user program touches the data transferred. The predicted file reread 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.

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 6 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 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. Alternately, if the 64KB transfer buffer is randomly relocated after every transfer, a similar performance drop is observed, as the buffer rarely gets reused at the same address, defeating the L2 caching effect. 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 6. 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.

Our final benchmark in the category of application-visible bulk data transfer primitives is the file read via mmap benchmark. This test examines how close user processes can come to attaining the raw memory read bandwidth when reading files that have been mapped into the process's address space via the `mmap()` system call. The test maps a 4MB region of an already-cached file into the user process's address space, then proceeds to read every byte. Since the 4MB region is significantly larger than the L2 cache size in all cases, it resides primarily in main memory, and thus we expect the available bandwidth to
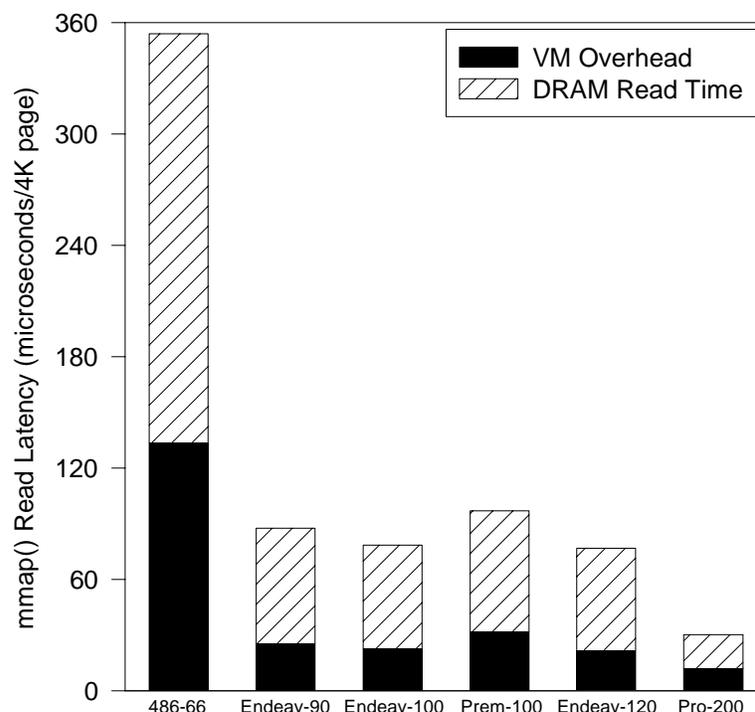
Figure 7: Mmap'd File Read Performance. This benchmark measures the bandwidth attainable using a memory-mapped interface to access files already resident in memory. The data is presented in terms of a latency—the amount of time needed to read one entire 4KB page as part of a larger 4MB read, in microseconds. The hatched part of each bar represents the time needed just to read the data; the solid section represents the time imposed by the virtual memory system (handling a fault, looking up the page, and installing TLB mappings). Both components of the total latency are heavily memory-system dependent, although the VM overhead has a small architectural component, due to hardware trap handling support.

be dominated by the main memory read bandwidth. However, there is another significant component that affects the attainable mmap'd-file read bandwidth: the overhead of the virtual memory (VM) hardware and software. For each page of data that the benchmark touches, the VM system takes a fault, even when the pages of the file are already cached in physical memory. Handling such a fault involves a protection domain crossing, executing some of the kernel's VM translation code, and entering a mapping into the user process's page tables, which reside in physical memory; use of the Pentium counters revealed that a TLB miss must also be serviced each time a new page is touched. Thus the time it takes to read a page of a mapped file decomposes into two sub-components: the main memory read bandwidth, which we have already seen in Figure 4, and the VM system's overhead. The measured results of the *hbench-OS* mmap-reread benchmark are presented in Figure 7 in terms of the time taken to read a (4KB) page; the component decomposition is indicated by stacked bars, with the main memory read time taken from the results in Figure 4.

As can be seen clearly from the figure, the main memory read time dominates the total mmap'd-file read latency, and thus the attainable bandwidth; the total time tracks closely with the memory read time. Interestingly, the VM overhead also tracks closely with the memory bandwidth, most likely due to the need to search page table structures and to write the hardware page tables. This is especially noticeable on the set of Pentium machines: the ratio of VM overhead to total time is approximately constant across this set of machines. There is a slight component of the VM overhead due to other architectural factors, though: the percentage of total time consisting of VM overhead is greater on both the *486-66* and *Pro-200* than on the Pentiums. Although the benchmark provides no data to support any additional conclusions, we hypothesize that the hardware support for the VM fault path and for the memory management unit updates is slightly better on the Pentium than on the *486-66*, and, similarly, that it is less optimal on the *Pro-200* than on the Pentiums (perhaps due to the need to flush or stall the deep CPU pipeline on faults).

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, TCP data transfer, and mmap'd file read 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.

2.3.2 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
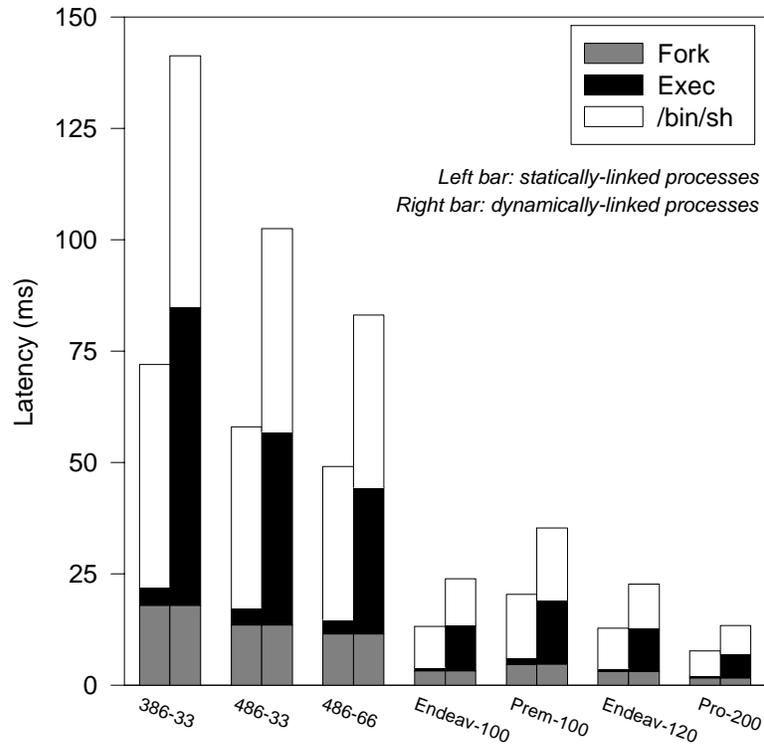
Figure 8: Process Creation Latencies. The total height of each bar represents the time to run "hello-world" via /bin/sh. The left bar in each group corresponds to the case where "hello-world" is statically linked; the right bar corresponds to a dynamically-linked "hello-world". The total process creation 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.

bars in Figure 8. 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.

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 8, 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 involved 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

39

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.

2.3.3 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, four times per accepted connection, according to our tracing results in Chapter 3). 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 9 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
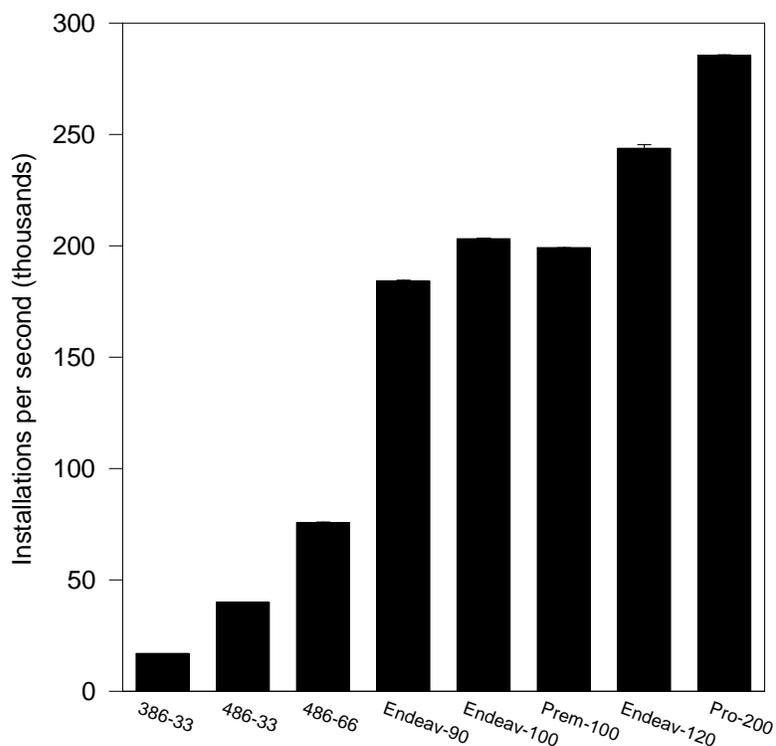
Figure 9: Signal Handler Installation. This graph plots the number of signal handler installations that each of our test machines can perform in one second. The results seem to scale with the CPU clock rate, but the factor of two performance difference between the *386-33* and the *486-33*, and a similar jump from the 486's to the Pentiums, suggests that the results of this test are determined by L1 cache performance.

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.

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 hardware feature that most influence 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.

2.4 Conclusions

The case study just presented demonstrates the success of our *hbench-OS* tools and accompanying decomposition methodology in constructing the detailed connections of the bottom half of the system performance hierarchy. In our sample 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 OS abstractions in order 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.

Finally, the results obtained in the case study themselves are of interest, and again highlight the effectiveness of the methodology used to obtain them. For example, 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][17][19], 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 *Xxpress*, eschews the advantages of EDO or synchronous DRAM for a large 1MB L2 cache since the larger cache produces higher SPECmark ratings [9]; our results suggest that to improve the performance of server applications that depend on OS-arbitrated bulk data transfer, Intel would have done better by engineering a higher-performance EDO DRAM-based system than by focusing on the caches.

However, despite the interesting conclusions we are able to draw about the effect of hardware design choices on the OS, we cannot use these conclusions as the sole basis for OS or hardware optimization, for we are still lacking the critical information relating operating system performance to patterns of application load. Without this data, most of the conclusions drawn here cannot be confirmed, for, without knowing which abstractions a given application exercises in the OS, we can only guess at which abstractions are important. For example, our conclusion that the Pentium Pro's cache coherency model is suboptimal on uniprocessors would not be very relevant if it turned out that the particular applications being run on the machine did not perform bulk data transfer. The following chapters will present techniques that reintroduce the application into the performance hierarchy and therefore allow verification of the types of conclusions drawn here.

Chapter 3

Extending the Performance Decomposition to User Applications

The previous chapter described tools and techniques that allowed us to characterize the bottom half of the system performance hierarchy described in Chapter 1. However, our ultimate goal is to understand OS performance in the context of performance paths that include the apex of the hierarchy (application performance). Thus, we need to extend the techniques of the last chapter to the user application level in order to construct the upper layers of the performance hierarchy. In doing so, we no longer have the luxury of the relatively-standard abstraction structuring that defined the lower half of the system performance hierarchy, for application designers are free to develop whatever abstractions they like on top of the standard[6] OS system call interface. Thus the methodologies that we describe are necessarily inexact, and provide only general guidelines for extending the performance decomposition through user-level applications. However, as the case study of the Apache web server in Section 3.3 will show, the methodology can produce surprisingly useful results.

---

6. Although the operating system designer has the flexibility to design his or her own abstractions as well, most modern operating systems must, by necessity, support the functionality of a POSIX-like system call interface, and therefore will be built upon an abstraction hierarchy similar in shape to that described in Chapter 2.

3.1 Methodology

From the point of view of the operating system, the only part of the application that matters is the layer that sits directly above the OS-exported abstractions in the performance hierarchy. This layer (along with its connections to the OS layer) includes the details of how the application exercises the operating system abstractions; from an operating systems perspective, any structure in the abstraction and performance hierarchies above that point is irrelevant. However, from a user's perspective, it is the *overall application performance* that is most important; if the application under consideration spends only a few percent of its time executing in the operating system, it is probably not an OS-dependent application, and any efforts spent tuning the operating system in response to it will not be worthwhile from the point-of-view of a user interacting with that application.

The first step in building the upper layers of the system performance hierarchy is thus to understand how (if at all) the application relies upon the operating system. To accomplish this, we again adopt the performance decomposition approach. Any application that relies on the operating system can be viewed in terms of the hierarchical structure of abstractions used by the performance decomposition technique. At the top of the *abstraction* hierarchy are the abstractions with which the user interacts—user interface elements, command-line functionality, or services provided. The performance of these abstractions is what the user sees directly, so they also correspond to the top of the application *performance* hierarchy. At the bottom of the application's performance hierarchy are the connections to the low-level resources of the system, including the computing hardware (for CPU-intensive applications) and the OS (for I/O-intensive applications). To determine how OS-dependent a given application is, we must understand the relationship between these two pieces of the hierarchy: how the user-visible abstractions decompose into OS and non-OS functionality.

This task of connecting high-level abstractions with low-level resources requires at least a rough knowledge of the internal structure of the application and its abstractions. If detailed knowledge of the application's control and data flow is available, this structure can be derived directly; we can just look at the type of work done by each user-visible abstraction in order to classify it as OS-dependent or non-OS-dependent. However, this is not a practical approach in general, for modern applications tend to be large, complex,

and proprietary, lacking source code and internal documentation. Thus we need a more general approach, one that can recover the operating system dependencies of user-visible primitives experimentally.

For this task, we turn to the type of experimental techniques developed in Chapter 2 in order to develop a more empirical approach. In Chapter 2, we isolated the relationship between the high-level OS primitives and the low-level hardware resources by varying the low-level resource to gauge the response in high-level performance. Although we could use a similar technique to discover the extent to which the application's user-visible abstractions depend on the operating system, there is a much more convenient methodology that is unique to the application domain. The key to this technique is the existence (in most modern systems) of tools that intercept and log the interactions between the application and the operating system, for example BSD's `ktrace`, Solaris's `truss`, or the Microsoft DDK's tracing DLLs for Windows NT. By tracing the application's use of the OS as each major user interface component, command-line option, or other functional unit of the application is exercised in turn, it is possible to directly isolate the dependence of each such user-visible primitive on the operating system. With this information, we bypass the need to construct a full application performance or abstraction hierarchy, for we already have the dependency detail that we need. Note, however, that this technique could be extended (by an interested application designer using similar performance decomposition techniques) to provide information about the performance and appropriateness of the internal abstraction structure of the application. We will not consider such an extension any further, since the level of detail it would provide, though potentially useful to the application designer, is not needed to study operating system performance.

Once a decomposition of the application's internal structure has been obtained by one of the techniques described above, the upper layers of the system performance hierarchy are complete. Assuming that the application decomposition indicates that at least part of the application's user-visible functionality (and thus performance) is OS-dependent, the next step is to tie together the application's part of the performance hierarchy with the operating system/hardware portion obtained via the techniques of Chapter 2. This is the

key step, for it is only when we assemble the complete performance hierarchy that we can understand and interpret OS performance in the context of realistic application load.

In carrying out this step, our goal is to be able to determine which OS primitives and abstractions are exercised by the application and to what degree. To do this, we use a more accurate version of the tracing technique described above: instead of merely recording the interactions between OS and application, we also record, for each system call handler or API function, how often it is called, the time spent in each invocation, and the parameters associated with each invocation. From this data, we can easily construct the pattern of load that the application imposes upon the operating system by using the call frequencies of and the parameters to each system/API call. For example, by looking at the calls to `read()`, we can determine how much data was transferred from the file system buffer cache or from network sockets (these two cases can be distinguished via the file descriptor used and previous calls to `open()` and `socket()`); this information additionally provides a connection with the OS-primitive performance measurements of Chapter 2, such as file reread bandwidth or TCP bandwidth. A more detailed discussion of how to use the performance hierarchy to connect the OS primitive measurements with application performance is the focus of Chapter 4.

Finally, the detailed traces of OS-application interaction can be used to identify operating system functionality that reduces application performance. In order to do this, the timing information gathered in the trace is used: the total time spent in each system call handler or API routine can be computed, and those routines that consume the most time (and thus affect performance the most) can be quickly identified. The information that comes from this timing data is invaluable for performance-tuning the operating system and the application, and can also indicate when certain OS abstractions are not appropriate for a given application.

3.2 Case Study: Developing Tools

We now shift our focus from general methodology to a specific example of its use. In this and following sections, we demonstrate how we applied the methodology described above to analyze the performance of NetBSD/i386 in the context of running the Apache web server [2]. Before plunging into a full discussion of the results that we obtained, we first

describe the tools that we constructed to simplify the process of gathering and obtaining the needed trace of interaction between the application and the operating system.

Since NetBSD is a BSD-derived system, it includes the `ktrace` program for monitoring system call activity (as described above). In its default incarnation, `ktrace` captures a wealth of detail about the interactions between the OS and traced application, including system call parameters and return values, I/O buffer contents, and pathname translations. However, we found NetBSD's `ktrace` facility lacking in two respects, and thus decided to augment it to make it more useful for our purposes, much like we did with the *lmbench* benchmarks in Chapter 2. The first deficiency that we needed to remedy in `ktrace` was in the way it handles system call trace records. An unmodified `ktrace` gathers two log records for each system call, one for the invocation and one for the return. Although both are timestamped, the elapsed time is not stored in either record, and thus, since the system calls are not sequence-numbered, there is no way to automatically correlate system call return records with their corresponding invocation records. This type of correlation is essential when tracing multiple processes simultaneously (e.g., the set of worker processes for a web server), for otherwise the records of each process arrive in an irregular interleave, making it very difficult to reconstruct the sequence and timing of the system call trace. Our solution to this problem involved a simple modification to the in-kernel ktrace facility: we added an "elapsed time" field to the system call return record, thus making it possible to match return records with invocation records. We also modified the `ktrace` facility to use cycle-accurate timestamps from the Pentium and Pentium Pro hardware timestamp counters rather than from the relatively low-precision in-kernel software clock.

These simple modifications were the only changes that we needed to make in the NetBSD kernel; the more significant problem with NetBSD's `ktrace` facility lay in the user-level data presentation engine, `kdump`. An unmodified `kdump` merely prints out the ktrace log records in the order they appear, making no attempt to correlate system call return records with their corresponding invocation records; the output format is not conducive to automated statistical analysis or abstracting. The second part of the methodology described in Section 3.1 requires summary statistics on a per-system-call handler

basis, and thus the uncorrelated time-sequence information produced by kdump is not useful.

Thus we implemented a new tool, ktranal, for processing ktrace log records. It operates directly on the raw ktrace records and can therefore bypass the unwieldy data presentation of kdump. Unlike kdump, ktranal uses the timestamps in the trace log to correlate system call invocations and returns, and internally maintains a per-process queue of unprocessed calls (to separate calls from multiple simultaneous processes). It supports several different "views" of the system call data. The first is a traditional kdump-like sequence of system calls, delivered to the user in time-sequence order, sorted by process, or for just one process of interest. Each displayed record contains the process name and PID, the call number and name, the start time of the call, the total time of the call, the arguments, and the return value. This view is useful for the first stage of the decomposition methodology described in Section 3.1, for it can give some idea of the traced application's operating system interaction, correlated in time-sequence with user actions.

The second view, a statistical summary mode, is most useful for the second stage of the application decomposition methodology. In this view, ktranal calculates and displays general statistics for each system call handler: how many times it was called, the total time spent in the OS while servicing that system call type, the average time spent per invocation, the minimum and maximum times spent, and the standard deviation of time spent. When sorted by the total time spent, this summary output highlights those system calls that dominate the OS component of the traced application's performance. This view can also be useful for gathering a "system call fingerprint" of a given application, for it concisely summarizes the application's interactions with the operating system.

The final view is useful when the summary output does not provide enough detail about a single system call, when the statistics indicate a non-normal distribution for a given system call, or when the user is only interested in one specific type of system call. This mode merely extracts the timing information for one system call number from the trace file and outputs it in time-sequence order. It can do this either for all traced processes or just one in particular, and additionally can output the call arguments along with the times. We found this output mode particularly useful when trying to differentiate between, for example, socket reads and file reads: by looking at the value of the file

descriptor argument along with the timings, we could easily distinguish the two sets of calls to the `read()` system call.

3.3 Case Study: The Apache Web Server

With our application performance decomposition and reworked `ktrace` tools in hand, we next turned to a real application in order to evaluate the effectiveness of the methodology and to illustrate what it can discover about operating system performance in the context of a real system running a real server application. For our testbed application and operating system platform, we chose the Apache web server (version 1.2b6) [2] running on NetBSD/i386 on the same Intel Pentium Pro-based machine referred to in Chapter 2 as *Pro-200*. We selected a web server as our example application because it is the canonical example of a widely-popular OS-dependent application; web servers are undoubtedly today's hottest server application, and as Table 1 on page 2 shows, can spend over 85% of their execution time running operating systems code. We chose the Apache web server in particular because it is a widely-used, high-performance, and non-proprietary web server. We also initially selected it because it includes source code, under the assumption that this would make the application decomposition easier; however, we did not ever have to resort to reading or understanding the Apache code while performing the decomposition. Finally, we selected the combination of NetBSD/i386 and the *Pro-200* as the target operating system/hardware platform so that we could build upon the OS performance decomposition detailed in Chapter 2.

3.3.1 Step 1: Decomposing Apache's Internal Structure

Recall that, despite the fact that our eventual goal is to draw conclusions about operating system performance, the first step in incorporating a realistic application load into the performance hierarchy analysis is to gather a rough characterization of the application's internal abstraction structure. We now describe how we applied the techniques for performing this characterization (described above) to Apache.

We began our decomposition with a two-layer structure derived from a general understanding of the tasks that a web server must perform during normal operation. We then refined this decomposition using data obtained via the empirical tracing techniques described in Section 3.1. We formed the rough decomposition from the top down: the

top-level abstraction of any server application is the service it provides to users, and for a web server like Apache, this uppermost abstraction is the action of serving a web page. In serving a typical file via the `http` protocol, Apache must do several things: it must accept the TCP connection from the client, read the client request over the TCP connection, read the requested page from the disk or buffer cache, send the page contents over the TCP connection, log the client request, and prepare for a new connection. These tasks, then, form the lower layer of abstraction in our rough internal decomposition of Apache. Note that each of the tasks is heavily dependent on I/O functionality provided by the operating system.

Before we proceeded to tie this set of lower-level abstractions to a notion of application load on the operating system, we first used our new `ktranal` tool to refine the rough characterization of abstractions and to determine exactly what operating system functionality is executed in tasks such as "log the client request". To do this, we gathered a trace of Apache as it responded to a single request for one HTML file (we chose the contents of Netscape's homepage (http://www.netscape.com) as it appeared on 22 March 1997). We ensured that the page's data was already present in the server's file system buffer cache, as that is the expected behavior for most servers that serve reasonably small (at most a few megabytes) file sets. Although the trace is too lengthy to reproduce here, it clearly showed the different phases of the server's operation once key system calls were isolated. For example, a call to `accept()` clearly indicates the phase of setting up the TCP connection; calls to `read()` and `write()` using the file descriptor returned by `accept()` represent reading the request from and writing the response to the TCP connection; calls to `stat()` and `open()`, and associated `read()`'s represent the retrieval of the page from the buffer cache. The trace was particularly illuminating during the logging stage of handling the connection (which was identified as the activity that occurred after the page had been sent over the TCP connection). The trace revealed that the logging stage can be decomposed into several lower-level tasks that are closer to the operating system interface (and thus easier to connect to operating system primitives): a DNS reverse-lookup request (involving sending and receiving UDP data), a write to the on-disk logfile, and several reads and writes to a temporary file apparently used for synchroniza-
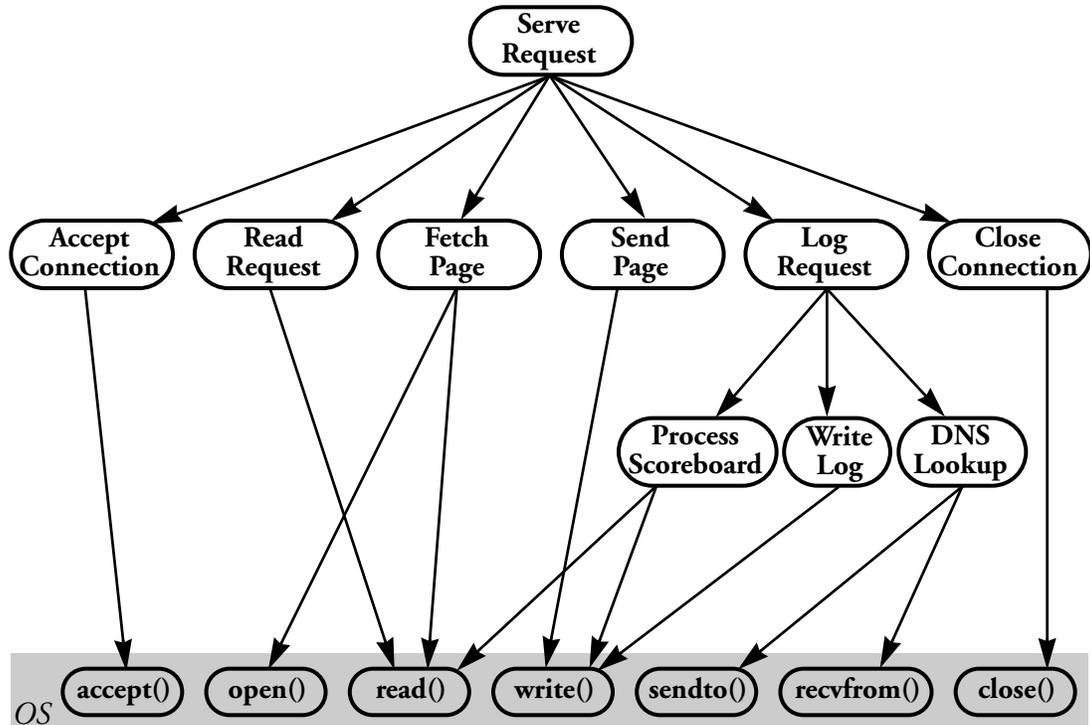
Figure 10: Abstraction-hierarchy Decomposition of the Apache Web Server. Apache's highest user-visible abstraction is the action of serving a request for a web page. This abstraction in turn relies on the entire hierarchy of abstractions depicted here. The structure of the hierarchy was obtained in rough form via an intuitive analysis, and was refined to the level of detail seen here via tracing techniques. Note that the lowest layer of abstractions (those contained in the gray box) are the OS primitives upon which Apache relies.

tion. The final decomposition that we constructed from the combination of the original two-level abstraction hierarchy and the refinements from tracing is presented in Figure 10.

3.3.2 Step 2: Connecting the Application and Operating System Hierarchies

The decomposition that we obtained of Apache's internal structure confirmed our intuition that Apache's abstractions (and therefore its performance) are heavily dependent upon the operating system. Thus we next proceeded to the second stage of the methodology described in Section 3.1 and attempted to characterize in detail both how Apache exercises the operating system and how the operating system affects Apache's performance. We began by using the summary mode of `ktranal` to gather timing statistics on the system calls present in a similar trace to that used in the earlier structure-characterization step; the trace used here was taken as Apache served the same Netscape HTML file from the OS buffer cache across the loopback interface. Thus the trace used here omits both the Ethernet transmission latency and DNS activity present in the earlier logs, for these are latencies introduced by factors that are both irrelevant to our investigation and

```
 NUM NAME          NCALLS   TOTALTIME     MINTIME     MAXTIME     AVGTIME      STDDEV
   3 read              19   0.0089698   0.0000195   0.0035422   0.0004721   0.0007929
   4 write             15   0.0029512   0.0000272   0.0014437   0.0001967   0.0003683
  93 select             1   0.0007875   0.0007875   0.0007875   0.0007875   0.0000000
   5 open               6   0.0004613   0.0000586   0.0001067   0.0000769   0.0000191
 199 lseek             18   0.0003132   0.0000147   0.0000340   0.0000174   0.0000044
  46 sigaction          4   0.0003033   0.0000221   0.0002175   0.0000758   0.0000946
  83 setitimer         12   0.0002992   0.0000178   0.0000339   0.0000249   0.0000047
 116 gettimeofd        11   0.0002510   0.0000185   0.0000404   0.0000228   0.0000064
 188 stat               1   0.0001546   0.0001546   0.0001546   0.0001546   0.0000000
   6 close              4   0.0001277   0.0000161   0.0000512   0.0000319   0.0000146
  20 getpid             7   0.0001124   0.0000143   0.0000174   0.0000161   0.0000011
  30 accept             1   0.0000798   0.0000798   0.0000798   0.0000798   0.0000000
 189 fstat              2   0.0000541   0.0000255   0.0000286   0.0000271   0.0000022
 117 getrusage          2   0.0000517   0.0000165   0.0000352   0.0000259   0.0000132
 134 shutdown           1   0.0000269   0.0000269   0.0000269   0.0000269   0.0000000
  32 getsocknam         1   0.0000267   0.0000267   0.0000267   0.0000267   0.0000000
 105 setsockopt         1   0.0000227   0.0000227   0.0000227   0.0000227   0.0000000
     TOTAL            106   0.0149931
```

Figure 11: Summary Statistics on System Calls made by Apache while Serving One Request. This figure depicts the summary output of the `ktranal` tool on a trace of Apache gathered while Apache served one copy of the Netscape home page to a client via the loopback interface. For each system call, the output includes the number of calls, total time (in seconds) spent in the call, the minimum, maximum, and average times spent in the call, and the standard deviation of the amount of time spent in the call. Notice in this particular example that calls to read and write dominate Apache's total execution time, accounting for 80% of the total in-kernel execution time.

---

beyond our control, namely network congestion and the performance of a remote machine.[7] The `ktranal` output is presented in Figure 11, and clearly illustrates those operating system features upon which Apache relies.

The first feature to notice is that the `read()` and `write()` system calls dominate the time spent in the kernel, accounting for 60% and 20% of the total 15 millisecond latency, respectively. However, the data indicates a large variance in per-call latency for both of these system calls. To examine why this was the case, we turned to `ktranal`'s ability to extract the parameters, including file descriptor and transfer size, for each invocation of `write()` and `read()`. `write()` turned out to be relatively straightforward: 88% of the write latency was due to socket writes, with one particularly slow transfer (probably due to full socket buffers necessitating a context switch to the client to read out the data); the remaining 12% was from writing very small amounts of data (less than 100 bytes) to various logs and status files. However, `read()` produced a much more interest-

---

7. In a production web-server environment, the DNS latency is an important factor in overall performance, as the last few bytes of the requested web page are not delivered to the client until after the lookup has been completed (this was revealed by the trace analysis). However, since we are primarily concerned with operating system performance and how the operating system affects server performance, any DNS latency (like random network congestion) is spurious to our considerations.

ing profile. We discovered that the largest single latency (accounting for about 40% of the total `read()` latency) was due to reading the request from the client over the TCP connection. More surprising, though, was that 49% of the total `read()` latency, nearly 4 milliseconds, resulted from 10 relatively-slow 50KB reads to a so-called "scoreboard" file, used for internal synchronization, coordination, and logging between the various worker processes that Apache uses to respond to requests. The remaining 11% of the latency was due to retrieving the page to be served from the buffer cache; although this was done in 7 separate requests, each was serviced quickly.

Thus, without even considering any of the other system calls made, we can already see the effectiveness of this analysis methodology. The results discussed above provide clear guidance on where the operating system affects application performance, and where tuning might be in order for both the operating system and application designers. From the application designer's viewpoint, the place to tune is in the manipulation of the "scoreboard": while the TCP transmissions and their associated latencies cannot be avoided by Apache, the server latency could be reduced by nearly one-third if Apache did not perform the frequent, large, and expensive "scoreboard" reads. The reason that Apache performs these reads, however, is because NetBSD does not export the abstraction that Apache expects (an `mmap()` interface that maintains coherency with ordinary file I/O), and as a result Apache is forced to revert to the less-efficient multiple-read technique. Our analysis methodology has thus also highlighted a deficiency in NetBSD's abstraction layer, the territory of NetBSD's OS designers. If the architects of NetBSD are concerned about web server performance, a good place to direct their tuning efforts would be toward modifying their mmap abstraction to support file system coherency. Alternatively, if the Apache designers are concerned with performance atop NetBSD, they might consider using an alternate method of synchronizing and controlling the slave processes that has less overhead than the large scoreboard reads.

Of course, there are many other areas where, according to the data in Figure 11, tuning of the operating system would be useful, for example in reducing TCP transmission latency by removing copies from the network stack, or improving the latency of opening a file by optimizing some of the VFS file-system abstraction-layer overhead. The key to the performance hierarchy methodology, however, is that it immediately pinpoints the critical

OS interactions that dominate application performance, preventing arbitrary and ineffective tuning. In the case of Apache, the methodology quickly revealed that the critical interaction revolves around the scoreboard file, and therefore we can conclude that it would be far less productive for either the application or operating system designer to optimize other interactions (such as opening files) until this critical interaction has been addressed.

3.4 Related Work: Understanding Application Performance

Because application performance is so critical to the end user's perception of overall system performance, much research has gone into understanding and optimizing for user application performance. However, a large percentage of this research, especially in the architecture community, has completely ignored the operating system, and has focused only on non-OS-dependent applications. The development of the SPECint and SPECfp benchmarks [24], for example, highlights the compute-intensive-workload focus of this community.

The small amount of research that has looked into understanding OS-dependent application performance divides into three different types. The first type of work uses tracing techniques to gather data on application performance; this type of work is the closest to the tracing methodology that we have proposed in Section 3.1, above. The most prominent example of this type of work is the Harvard/University of Washington Etch system [6], which uses binary rewriting to dynamically add instrumentation code to Windows applications. This instrumentation code can then gather performance details during the application's regular execution. Although Etch appears to be mainly used for architecture-based analysis (such as cache organization evaluation), it appears general enough that it could be used as a replacement for the type of tracing tool that we describe above. Thus Etch and similar projects do not replace the performance decomposition methodology that we have developed, although they could be useful as a replacement for the `ktrace` tool in commodity systems.

The second group of research focuses on trying to understand the interaction of application and OS performance by experimentally measuring the application as it runs or by experimentally studying how the running operating system reacts to the patterns of appli-

cation load. The primary method by which this is achieved is through some sort of profiling. At the most basic level, tools such as *gprof* [8] allow user-level profiling of applications. At most, these tools can indicate what application abstractions are bottlenecks or especially slow; they cannot determine how much that slowness depends on the operating system's performance, and thus are not useful in the type of decompositional analysis that we are performing. More useful profiling results can be obtained by moving the profiling from application level to kernel level: tools such as the BSD *kgmon* kernel profiling system [13] record the time spent in each kernel function as well as the kernel call graph traversed during execution of a given application. Kernel profiling data makes it easy to determine the overall OS dependency of a given application: if the amount of time spent in kernel functions in total is a significant fraction of the application's execution time, then the application is most likely OS-dependent. However, kernel profiling has several disadvantages. First, it does not charge idle time to user processes: if the process must wait on disk or network I/O, or is responding to an interrupt, this time does not show up as being spent inside a specific kernel function, and thus cannot be associated with the user application's actions. Additionally, kernel profiling provides only aggregate performance data (a snapshot of total kernel use while the profiler was running): it cannot distinguish kernel time spent servicing processes other than the desired application process. More seriously, it cannot provide the time series information that is essential for correlating different user application functionality with kernel usage; with a trace-based methodology such as the one presented in Section 3.1, it is possible to correlate each of a sequence of user actions with the application's corresponding kernel usage. Some of these problems are solved by using process-tagged kernel profilers such as *pkprof* [16]; however, even these profilers still provide only aggregate data and cannot supply profiling data over time.

Yet another profiling approach is to avoid any special kernel profiling tools at all and to instead use such hardware features as the Pentium's performance monitoring counters to gather information on a running system executing a particular application/OS combination. This was the technique used by Chen et al. in their study of commodity operating system performance [4]; unfortunately, such techniques provide even less information than traditional kernel profiling, and thus also cannot be used for the decompositional performance analysis approach.

The third group of research into OS-dependent application performance eschews experimentation entirely, and moves into a simulated domain. The prime example of this work is in the Stanford SimOS project [20], in which a detailed machine simulator is used to boot a real copy of the operating system and application under test. The simulator supports tools for gathering general performance statistics as well as recording and tracking events and performance characteristics of interest. This simulation approach probably provides the best way to determine the operating system dependencies of an application, for it is possible to observe in complete detail each interaction between the application and operating system. However, simulation is not a feasible approach in general, for it requires that a complete, correct simulated environment be built for each hardware platform of interest; it also requires some modification to the operating system under test. The biggest drawback is that the simulator does not run in real time, and thus cannot be used to understand applications that interact with non-simulated entities (such as network servers); with the trace-based analysis that we propose, it is possible to gather performance data even from running applications that perform all of their normal functions unchanged.

3.5 Conclusions

The case study of the Apache web server presented in this chapter has yet again underscored the critical need to consider both the operating system and application when attempting to analyze the performance of either. The goal of performance measurement is to either evaluate the user-visible performance of a system or to indicate where the performance bottlenecks in a system can be found. Since real systems include both applications and operating systems acting in concert, neither fixed application workloads (like SPEC) nor simple operating system profiles (like those created by the *hbench-OS* microbenchmarks in Chapter 2) are sufficient to satisfy the goals of performance measurement. In the case of Apache, neither alternate method would have indicated the problem with the mmap abstraction that was revealed by our combined analysis methodology; without this information, neither would have produced an accurate evaluation of the overall system performance; and thus neither would have been able to direct the OS or application architect to the spot where tuning will be most useful.

Chapter 4

Distilling the Detail: Performance at the OS-Application Abstraction Boundary

In the previous two chapters, we have focused on the intricacies of building detailed performance hierarchies and interpreting their structure in order to analyze the performance of operating systems under application load. In this chapter, we take a step back and consider how to distill some of the complexity and detail contained in the performance hierarchy into simpler characterizations and metrics. This may seem counter-intuitive at first, since we have just devoted many pages to describing techniques designed to extract and preserve the maximum amount of detail possible out of system interactions and performance. However, the amount of detail contained in a full system performance hierarchy that includes everything from the hardware up through the application is overwhelming, and is probably of interest only to the most sophisticated operating system designers. Most less-sophisticated consumers of benchmarks, such as system administrators, application developers, and end users are not interested in performance-tuning the operating system and thus would probably prefer simple, small, and meaningful metrics to represent operating system performance. Thus our goal now is to try to preserve some of the power of the decompositional method of performance evaluation while reducing the complexity of its results to a level useful to such less-sophisticated consumers.

This is not a simple task. We have already seen the inherent flaws in OS measurement methodologies that deliver only a single inflexible metric, for they do not take into

account the fact that the operating system is, for the most part, a reactive entity whose performance depends almost entirely on the pattern of application load imposed on it. Since it is these patterns of application load that define OS performance, any simple metric that we develop must either retain enough detail to be adaptable to different application loads, or else must irrevocably incorporate a specific application's workload into the metric. The latter option is not very desirable, since there already exists an (uninteresting) way of measuring the performance of a fixed combination of a specific application and OS: one can just run the application atop the OS and measure its user-visible performance directly. Thus we adopt the former option, and focus on the development of a methodology that permits independent characterization of the operating system and application in a way that allows the independently-derived characterizations (obtained via the *hbench-OS* benchmarks of Chapter 2 and the tracing techniques of Chapter 3) to be recombined into a single performance-predicting metric.

4.1 Methodology

The approach that we have taken to accomplish this goal is based on one that revolutionized the reporting of performance in the architecture community several years ago. Before the RISC revolution, the architecture community faced a set of problems similar to those we are considering here: hardware performance at that time was measured via *ad hoc* benchmarks that produced results that did not reflect true application demands on the hardware. The prime example of such benchmarks is the *peak MIPS* rating, which merely measures the theoretical maximum number of instructions that can be executed by a processor in a second. The problems with MIPS are twofold: first, MIPS can be measured by timing simple compute instructions only, and thus it is possible to obtain higher MIPS values than would be seen with real application code that interleaved compute instructions with memory accesses; second, the amount of work done by an instruction can vary greatly between architectures (e.g. CISC vs. RISC), and therefore it is usually impossible to compare architectures based on MIPS ratings. With the coming of the RISC revolution in 1980, these flaws of MIPS were highlighted, and the community began to search for more useful metrics.

The solution formalized by Hennessy and Patterson revolves around CPI (Cycles Per Instruction), an application-driven performance metric that is independent of the complexity of the instruction set [18]. CPI is computed by measuring, for each type of instruction executed by a specific application, the average number of cycles spent executing that particular instruction (thus, for example, instructions that access memory might accrue more cycles than ALU-manipulating instructions; similarly, CISC instructions would accumulate more cycles than RISC instructions). The numbers of cycles for each instruction are then weighted by the frequencies with which those instructions are executed by the application and summed, producing a single number ("CPI") that represents the average time spent per instruction for the application used. Thus, in the process of calculating CPI, the detailed interactions between the hardware and the test workload are revealed (via the instruction timings and frequencies), yet this detail can be distilled into the single number (CPI) that the end-user seeks.

The essential methodology behind the calculation of CPI consists of three stages. The first is to identify and measure the performance of the primitives exported by the hardware (for CPI, the cycle times of each instruction supported by the hardware). In effect, this stage produces a detailed characterization of the hardware's performance as a vector of instruction cycle timings. By itself, however, this characterization is not very useful, for it does not reflect the relative importance of each of the instructions. To see this, consider the oft-cited VAX instruction that evaluates a polynomial function: this instruction has a very high CPI since it must do so much work. If it is never used by an application, though, it will not contribute to the overall CPI value for the application/hardware pairing. The second stage of the CPI methodology addresses this problem by producing a characterization of the application's use of the hardware primitives measured in the first stage. For CPI, this is relatively trivial, for the characterization consists of just the frequencies at which the application executes each instruction. Finally, the last stage of the CPI methodology combines the two characterizations obtained in the earlier stages into a single performance metric that represents the profiled application's performance on the characterized hardware platform. Again, in the case of CPI, this not very difficult, since the instruction frequencies from the application characterization can simply be used as the

weights in a weighted average of the cycle counts obtained in the hardware characterization.

The goals of the methodology used to compute CPI are very similar to the goals that we wanted to achieve in OS measurement, most notably in the way that CPI allows independent performance characterization of the hardware and application in such a way that the independent characterizations can later be synthesized and recombined to produce a useful metric that can be compared across applications and architectures in turn. Thus we applied the same techniques to the operating system domain, but with the focus on interactions at the OS-application boundary rather than on the application-hardware boundary. Note that, although its methodology provides a useful blueprint, the specific metric of CPI by itself does not provide a solution to the problems of OS measurement, since it is designed to measure the performance of non-OS dependent applications. In fact, CPI either ignores the OS entirely, or else treats the OS as part of the application.

Thus, in implementing an OS-domain equivalent of CPI, we had to accomplish three tasks: identifying a set of general operating system primitives that characterized the space of OS performance, developing a means of decomposing application performance into its use of such primitives, and creating a technique for combining the OS and application characterizations into a single, useful metric. In the process of developing the performance decomposition methodology, however, we had already done much of the work needed to accomplish the first two of these three goals: *hbench-OS*, described in Chapter 2, already provided us with a set of tests designed to characterize the OS primitive abstractions exported to user applications, and the application profiling techniques of Chapter 3 provided a way of profiling an application's use of such abstractions. We were thus left with taking the detailed performance hierarchy produced by the techniques in these earlier chapters and transforming it into simple characterization vectors for the OS and the application.

In order to do this, our first task was to identify a basis for the space from which the characterization vectors could be drawn. With CPI, meeting this goal was trivial, as the target machine's instruction set itself provided an easily-identified and exhaustive set of basis elements. With the operating system, however, the task becomes much more difficult, for the OS primitives used by applications are not as well-defined and are thus harder

to distill into discrete components. To solve this distillation problem, we appealed to the system performance hierarchy model. By taking a detailed performance hierarchy constructed using the techniques of the previous two chapters, then slicing it horizontally at the boundary between the application and the operating system, we found it easy to identify the basis elements for our characterization vectors: they comprised the layer of OS abstractions that lay just below this slice (i.e., those at the top of the OS part of the performance hierarchy), as all interactions between the operating system and the application pass through this layer of abstraction. Thus each high-level OS primitive becomes a component of the characterization basis; the OS characterization vector is then made up of coefficients representing the performance of each such primitive, while the application characterization vector is composed of coefficients that specify how the application uses each primitive. Note that the mapping of basis elements to OS primitives need not be one-to-one, as a given primitive may have several associated coefficients corresponding to different cache behavior or data set size (i.e., the file and TCP bandwidth primitives perform differently depending on the size and location of the transfer buffer, so separate coefficients are needed for the cases when the buffer is in the L1 cache, the L2 cache, or in main memory, and for various typical buffer sizes as 4KB, 8KB, and 64KB).

With the basis for our characterization vectors established, the next task was to define the precise meaning of the coefficients of the characterization vectors. The OS characterization vector represents the performance of each of a set of varied OS primitives and high-level abstractions, so the natural choice for the values of these coefficients was the set of results obtained by using *hbench-OS* to profile the operating system being characterized; since the *hbench-OS* tests expose the structure of the system performance hierarchy at the OS-application boundary, their results correspond directly to the performance of the elements of the characterization basis. However, the *hbench-OS* results include a mixture of bandwidths and latencies, and thus we found it necessary to develop one more normalization step to make the coefficients self-consistent. To do this, we chose a simple approach: all results were converted into latency metrics, either of the form milliseconds per byte (for data-transfer primitives) or milliseconds per invocation (for discrete primitives like process creation), before being used as the coefficients for the OS characterization vector.

Before we proceed to discuss the form of the coefficients of the application characterization vector, note that in creating the OS vector we have achieved one of our most important goals in creating an operating system performance metric: we have not tied the vector down to any specific application load. In fact, the vector is essentially meaningless without a corresponding application characterization, for it merely reports the performance of a spectrum of OS functionality without distinguishing any particular functionality as most important. This is similar to what we saw in the CPI example: although certain elements of the hardware characterization vector may have had high individual CPIs (e.g., the VAX polynomial-evaluation instruction), it was impossible to tell how they would impact the overall CPI without first seeing if the application characterization vector indicated a reliance on those elements.

The application characterization vector is thus the key to unlocking the performance secrets hidden within the OS characterization, for it contains the crucial ranking of the importance of each OS abstraction to the application's performance. The coefficients of the application vector have the task of encoding this ranking, and thus must be representative of the application's performance demands. As a result, the method that we chose to create these coefficients is heavily based on the tools and techniques for accurate tracing of realistic, live workloads described in Chapter 3. Since the coefficients of the OS characterization vector are in units of time per byte or per invocation, a natural corresponding form for the application characterization vector's coefficients exists: for each data-movement primitive in the basis, the coefficient is the number of bytes transferred by the application, and for each remaining discrete primitive, the coefficient is the number of times the primitive is invoked. This information can be easily obtained from the output of the `ktranal` tool described in Chapter 3. The application vector is then just a compilation of this information, consisting of a string of coefficients representing the application's use of each OS primitive.

With the application and OS characterization vectors created as described above, producing the single performance metric involves simply taking the dot product of the two vectors. The units of the coefficients were carefully selected to multiply into an absolute unit of time, and so as the coefficients are multiplied and added, the relative importance of each OS primitive to the application is automatically incorporated with the perfor-

mance of that primitive into a share of the resultant metric that is proportional to how much of a role that primitive plays in the application's performance. Thus the resultant single metric, represented as a latency in milliseconds, is inversely proportional to the true performance (throughput) of the application on the system characterized in the OS vector.

4.2 Analysis of Methodology and Potential Future Research Directions

Although the methodology described above succeeds as an apparatus for distilling the detail of the application performance hierarchy while still preserving the interplay between operating system and application, our experience with it is still too limited for us to conclude that it captures the entire set of performance characteristics of either the application or the operating system. In fact, there are several obvious areas in which the methodology is still incomplete. Probably the largest gap in its coverage of OS performance is that it does not take into account contention between multiple applications: if a given application is characterized on a system where it is the only thing running, it is not clear that anything useful can be said about the application running on another system that is already heavily loaded by other applications. One possible solution to this problem would be to add the two application characterization vectors before taking the dot product with the OS characterization vector; since the coefficients in the application vector are additive, this should produce a meaningful result. However, this technique ignores issues like cache contention and context switching overhead that occur when multiple applications are being run. Since these overheads are hard to encode in the vector format we describe above, it is difficult to incorporate them into our analysis framework, and more research will be needed in order to do so. In a similar vein, the characterization technique described herein does not scale well to machines with multiple processors: since *hbench-OS* characterizes the system as seen by one process, it does not capture potential performance improvements that come from per-processor caches and page tables. A promising solution to this problem involves running multiple copies of the *hbench-OS* tests in parallel (one for each CPU) to gauge the effective total performance of the system; we have obtained promising results by using this technique on a 2-CPU SPARCstation 20, but more research is still needed to confirm the efficacy of the technique.

The remaining gap in Section 4.1's distillation methodology concerns the application characterization. Although the *hbench-OS* primitives cover the major abstractions exported by the operating system, there are some application-OS interactions that do not correspond to any *hbench-OS* tests. For example, such system calls as `getrusage()` or `setsockopt()` do not have equivalent *hbench-OS* primitive measurements. It is possible, however, to use *hbench-OS*'s null system call benchmark to provide a lower bound on the latency of such calls. More detailed analysis of the actions performed by such calls is needed in order to expand *hbench-OS* to include these types of application-operating system interactions.

4.3 Case Study: Predicting the Performance of the Apache Web Server

Despite all of its gaps, the characterization methodology of Section 4.1 is powerful enough that even a first-order approximation is sufficient in some cases to capture the important performance characteristics of an application/OS pairing. In this section, we demonstrate such a case by presenting the results of using the techniques of Section 4.1 to characterize the Apache web server and to predict its performance across a range of different hardware/OS platforms.

The first task in performing this analysis was to produce the characterization vector corresponding to Apache's resource demands. For clarity in this example, we will work with a set of basis elements that includes just those OS abstractions used by Apache; the full basis space is too large to represent conveniently here. In order to determine the contents of this reduced basis set, we first needed to relate the abstractions that are important to Apache's performance to the operating system primitives measured by *hbench-OS*. We used the data from the trace in Chapter 3 (from serving a cached copy of the Netscape home page) to support this analysis. As we saw in Chapter 3, Apache's performance is dominated by reads and writes (together they account for 80% of Apache's execution time on the *Pro-200*). We also saw that these reads and writes decomposed into subcategories: TCP socket reads and writes, and file reads and writes. Under the assumption that every byte transferred via the TCP socket must have passed entirely through the TCP stack, we associated the socket reads and writes with the TCP loopback bandwidth primitive of *hbench-OS*; in particular, since all the data transfers used buffers of a size between 4K and

8K, we chose the version of *hbench-OS*'s `bw_tcp` benchmark that used 4K buffers. The application characterization coefficient for this element thus became the number of bytes transferred in total, specifically 34,947 in the case of serving the Netscape home page.

We followed a similar procedure for the file reads. In serving just one copy of the Netscape home page, Apache generates 518,184 bytes of file read traffic, the bulk of which consists of reading the scoreboard file (about 32K of that traffic is due to reading the actual HTML page from the buffer cache). All of these reads can reasonably be assumed to be serviced from the file system buffer cache, and are performed in transfer sizes that average a bit more than 32KB. We thus selected *hbench-OS*'s cached file reread test (`bw_file_rd2`) with 32KB transfer buffers as the next basis element for Apache's characterization; the associated coefficient of the application characterization vector was the number of bytes transferred, or 518,184.

Connecting file writes to an *hbench-OS* primitive illustrated one of the gaps in *hbench-OS*'s coverage of OS primitives; there is no component of *hbench-OS* that measures cached file writes directly. However, we were able to improvise a still-useful metric by using a similar method to that used to generate the predicted file read measurements in Section 2.3.1: since a file write involves a memory write (to fill the buffer) plus a copy (to put the buffer in the buffer cache), we approximated file write with the level-two-cached memory bandwidth primitives of *hbench-OS*; we used the number of bytes written (1533) as the coefficient of both the L2-write and L2-copy bandwidth basis elements.

We proceeded in a similar manner to transform the other interactions revealed by the `ktranal` output displayed in Figure 11 in Chapter 3, where possible. Since we were primarily interested in seeing how just a rough first-order approximation to a characterization of Apache would perform, we did not go to great pains to resolve all of the less-important interactions into *hbench-OS* primitives. Some we could not resolve at all, for example the sole call to `select()`: the latency introduced by this call is dependent on the scheduling of the driver program that generated the trace and not on any OS abstraction. Thus, other than the call to `accept()`, which we resolved to TCP connection latency, and the calls to `sigaction()`, which were resolved to signal handler installation latency, we merely mapped the remaining system calls to the null system call benchmark to provide a lower bound on their performance impact. The final set of basis
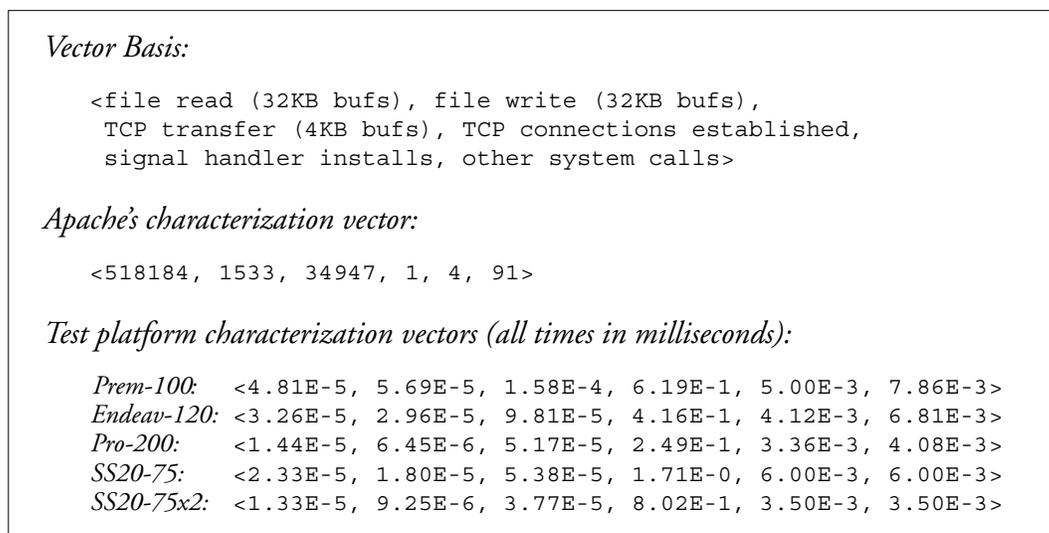
Vector Basis:

```
<file read (32KB bufs), file write (32KB bufs),
 TCP transfer (4KB bufs), TCP connections established,
 signal handler installs, other system calls>
```

Apache's characterization vector:

```
<518184, 1533, 34947, 1, 4, 91>
```

Test platform characterization vectors (all times in milliseconds):

```
Prem-100:   <4.81E-5, 5.69E-5, 1.58E-4, 6.19E-1, 5.00E-3, 7.86E-3>
Endeav-120: <3.26E-5, 2.96E-5, 9.81E-5, 4.16E-1, 4.12E-3, 6.81E-3>
Pro-200:    <1.44E-5, 6.45E-6, 5.17E-5, 2.49E-1, 3.36E-3, 4.08E-3>
SS20-75:    <2.33E-5, 1.80E-5, 5.38E-5, 1.71E-0, 6.00E-3, 6.00E-3>
SS20-75x2:  <1.33E-5, 9.25E-6, 3.77E-5, 8.02E-1, 3.50E-3, 3.50E-3>
```

Figure 12: Characterization Vectors for Apache and for the Test Platforms. The basis declared at the top of this figure defines the components of each of the subsequently-listed characterization vectors. The coefficients of the Apache characterization vector represent Apache's use of each OS primitive in the basis, either in bytes transferred via the primitive or in the number of invocations of the primitive. The coefficients of each platform's characterization vector represent the performance of each primitive on that platform as a latency metric, in either milliseconds per byte (for data-transfer primitives) or milliseconds per invocation (for discrete primitives).

elements that we used and the resultant Apache characterization vector are listed in Figure 12.

With this approximation to Apache's true performance characterization vector in hand, we next turned to gathering characterization vectors for several operating system and hardware platforms. For the set of OS/hardware combinations, we chose the *Prem-100*, the *Endeav-120*, and the *Pro-200* from the results in Chapter 2, and additionally, as a challenge to the methodology, added what we will refer to as the *SS20-75*, a Sun SPARC-station-20 with a 75 MHz SuperSPARC-II processor, 1MB second-level cache, and 128MB of DRAM running Sun's Solaris 2.4 UNIX operating system; we characterized this machine with the same set of *hbench-OS* benchmarks that were used to measure the Intel machines under NetBSD.[8] The reason for including the *SS20-75* was to see if the characterization methodology scaled well across major hardware architecture and OS differences (the SPARC and Intel architectures differ greatly, as do Solaris and NetBSD).

Building the characterization vectors for the OS/hardware support platforms was relatively trivial given the work already done in selecting the basis elements for the application

---

8. Note that we compiled Apache for Solaris using the same options as for NetBSD; in particular, we did not alter the scoreboard synchronization policy.

performance vector. As specified by the methodology, we merely took the *hbench-OS* results for each platform, normalized them as described in Section 4.1 to units of milliseconds per byte or milliseconds per iteration, and arranged them in the proper order to form the desired characterization vectors. The characterization vectors for each of the systems considered are also given in Figure 12.

At this point, with both the Apache characterization vector and those for the underlying OS/hardware platforms in hand, only the last step of the methodology, taking the vector dot-product to obtain the desired single performance metric, remained. However, before discussing the actual results obtained, let us take a step back and consider what the calculated metrics actually mean in the context of this example. Taking the dot product of an application vector and an OS vector produces a number representing a latency in milliseconds. With a completely OS-dependent application and a perfect mapping from application usage patterns to *hbench-OS* primitives, this latency has an absolute meaning: it should precisely predict the application's absolute, user-measurable performance. In our case, however, with a heavily (but not completely) OS-dependent application and only an approximate mapping from the application's resource demands (as visible in the trace results) to *hbench-OS* primitives, we cannot expect the latency result to measure any real, absolute time; in particular, it is *not* the time spent by Apache in servicing a request for the Netscape home page, although it should be of the same order of magnitude. The reason why this is the case is that, in establishing the vectors, the exact detail of the true latencies and timings was lost in shifting from real traces to abstractions. As an example, consider the transfer of data over the TCP connection. We used the amount of data transferred by the server as the coefficient for a basis element that measured total TCP throughput. Thus, the contribution of this basis element to the final latency produced by the dot product is the amount of time it should take to transfer a specified amount of data over the TCP loopback interface. This is not the same amount of time it takes the server to enqueue the data for transmission, which is the time being measured by the tracing methodology of Chapter 3. However, it is time that contributes to the latency that the client sees in making a server request, and so it does impact user-visible performance. Thus the time is still meaningful; it just pushes the calculated latency metric away from a direct correspondence to the true request latency.

However, the calculated metric still provides a good means for *relative* comparison: it captures the important performance dependencies of the application and thus can provide a way of gauging or ranking the relative performance of several systems, even if it cannot predict the absolute performance. This is one of the critical goals of performance evaluation, and if our methodology successfully produces relative rankings of different systems, it has met this goal. Thus it is the ability to predict relative performance that we use in evaluating the success of predictions based on the calculated metrics.

In order to perform this evaluation, we needed a standard by which to judge the predictions of the characterization-derived metrics. Thus we had to both find some directly-measurable metric of Apache's performance, and subsequently use it to establish a standard rank order of the different OS and hardware platforms that was correct from the point-of-view of user-perceived performance. We chose to use a simple throughput metric for this task: we measured the average number of requests per second that Apache was able to service in processing two different client loads. The first was a simple workload that consisted of retrieving the Netscape home page 5,000 times in sequence; this was our "control" workload, as it precisely duplicated the type of request used when gathering the application trace in Chapter 3. The second was a more complex workload that consisted of replaying a subset of server logs from an NCSA server; this workload generated over 15,000 requests covering a mixed fileset of approximately 962KB (which is large enough to be realistic, but still small enough to fit in the buffer cache of all our test systems).

We measured Apache serving these workloads on all of our testbed platforms with all data transfer done via the loopback interface; the measured performance results are presented in Table 4 along with the calculated latency metrics obtained by taking the dot product of the Apache characterization vector with each of the testbed platforms's OS/hardware characterization vector. As can be plainly seen, the calculated metrics rank the machines in exactly the same order of relative performance as do the experimental measurements, for both workloads (recall that the calculated metric represents a latency, the inverse unit of the measured throughputs). In fact, when scaled by a factor chosen to best approximate the average measured performance, the calculated metrics predict the absolute measured performance on the Netscape trace (in requests served per second) to within about ±16%. This is an encouraging result since, due to our crude application of

69

Test Platform	Calculated Latency (ms)	Measured Throughput (requests/sec)	
		Netscape Workload	NCSA Workload
Prem-100	31.88	22.71	25.93
Endeav-120	21.43	38.35	44.03
SS20-75	16.26	44.78	47.59
Pro-200	10.14	62.90	64.11
SS20-75x2	9.33	63.82	66.72

Table 4: Comparison of Measured and Predicted Apache Performance Rankings. The measured throughput numbers represent the average number of requests serviced per second by the Apache web server on each test platform; they are averages over several thousand requests. The calculated latency figures were obtained by taking the dot product of each platform's characterization vector with Apache's characterization vector, and represent predictions of each platform's performance. Notice that the calculated latency figures rank the systems in the same performance order as the measured throughput figures (recall that latency is inversely proportional to throughput, so lower latency implies higher throughput).

the characterization methodology, we did not even expect the calculated metrics to approximate the absolute performance. Additionally, this prediction holds (with a slightly greater margin of error) on the mixed NCSA workload, despite the fact that the original characterization was based on the Netscape trace. Even more promising is the fact that the SPARC/Solaris machine (*SS20-75*) is placed correctly amongst its Intel/NetBSD brethren on both workloads; this is a particularly interesting result, as it shows that the characterization methodology used is powerful enough to allow for *meaningful* cross-platform comparisons, something that is almost entirely unattainable when measurements are performed only via microbenchmark suites such as *lmbench*.

As a final experiment, we attempted to extend our methodology to the multi-CPU case to see if its cross-platform predicting power held there as well: we installed a second 75 MHz SuperSPARC-II CPU in the SPARCstation-20, and ran two copies of the *hbench-OS* tests in parallel, adding bandwidths and taking half the harmonic mean of the latencies to obtain a characterization vector representing the aggregate performance of the system. The calculated performance metric for Apache on this configuration (which we denote *SS20-75x2*) came out to be 9.33 ms; the measured user-visible performance on the two workloads was 63.82 requests/second (Netscape) and 66.72 requests/second (NCSA). Although this result is less successful at predicting absolute performance, it still correctly predicts the relative position of the dual-CPU machine amongst its single-CPU companions, even though the machine's characterization was relatively crude.

Thus, in the case of Apache, our characterization methodology was successful, as it met one of the primary goals of system performance evaluation: it provided a way to gauge the relative performance of different operating system/hardware platforms on a given application. Although we varied the underlying platform by changing the OS and the hardware, our technique is equally applicable for changes within the OS on the same platform, and therefore it also provides a way for the operating system designer to evaluate changes to the OS (or for the hardware designer to evaluate hardware changes) to see if they help real applications. Additionally, the success of the methodology here despite its rather inexact application implies that the technique of characterization and the resultant computed metrics are capable of capturing the important performance characteristics of an application/operating system pairing. The fact that we could take a characterization of Apache obtained on a single-CPU Pentium Pro running BSD and subsequently use it to correctly predict Apache's relative performance on both the single- and dual-CPU SPARC/Solaris machines highlights the resiliency and power of our methodology. In addition, the technique's success in meeting the critical goal of performance evaluation described above again underscores the fact that even a rough gauge of operating system performance *in the context of real application load* is more useful than just a set of measurements of OS primitive performance, for example as obtained via *lmbench*.

4.4 Related Work: Application Performance Characterization and Prediction

The methodology that we have developed for producing application-specific metrics for system performance evaluation and prediction is not unique. As we have already seen, it closely follows the standard methodological guidelines laid down by CPI. Although this is the first time that such a methodology has been applied to the abstraction boundary between the operating system and the application, other researchers have used the characterization methodology at other abstraction boundaries in the computer system. The primary example of this is in the work of Saavedra and Smith, who have applied the characterization methodology to the boundary between CPU-dependent scientific applications and the low-level machine hardware (essentially ignoring or bypassing the OS) [21] [22]. They developed a characterization model based on Fortran primitives: the low-level hardware is characterized by measuring each primitive's performance, and the result-

ant measurements are combined into a characterization vector in much the same manner as we do for the operating system primitive measurements. Similarly, the (Fortran-based) application is characterized by a vector indicating its use of each primitive, and the end performance metric is obtained via a dot product of an application vector with a machine vector. Saavedra's work is thus quite similar in principle to what we have done, although in practice it is more complimentary, as it serves an entirely different class of applications than the OS-dependent applications we are considering.

Chapter 5

Conclusion

Throughout this thesis, we have presented methodologies and techniques designed to deliver accurate, detailed performance characterizations that reflect and distill real-world workloads and conditions. The only question that remains is whether we have succeeded in achieving the goals laid out in Chapter 1; from the results obtained in the case studies throughout, the answer is clearly yes.

The performance decomposition approach and the performance hierarchy model upon which it is based produce a system performance characterization with enough detail to satisfy even the most demanding performance analysis needs of a sophisticated operating system architect. The constructed performance hierarchy incorporates performance dependencies at all levels of the system, allowing the architect to compare, evaluate, or tune the system at any level of abstraction, from the physical hardware to the OS implementation to the OS abstraction layer to the application itself.

At the same time, the distillation techniques of Chapter 4 allow all of this detail to be reduced to a set of simple application and operating system characterization vectors that, when combined via a basic dot-product operation, produce a single, application-specific metric for evaluating operating system performance. This metric satisfies the requirements we set out in Chapter 1 as well. It is simple enough to be easily constructed and understood by less-sophisticated consumers of benchmarks, yet incorporates enough knowledge about the application workload and the operating system's abstractions to provide a mean-

ingful measurement of performance. Like the full, detailed hierarchy, the characterization vectors and resultant single metric satisfy all of the potential demands for performance analysis: system evaluation, comparison, and optimization. Finally, although the single metric produced is workload-specific, the fact that it can be trivially constructed from the application and operating system characterization vectors means that it is easily adaptable to any workload of interest, unlike such static metrics as SPEC or the *lmbench* micro-benchmark results.

The case studies of NetBSD/i386 and the Apache web server further illustrate the practical usefulness of the decompositional performance analysis approach. In construct-ing the performance hierarchy for the x86-NetBSD-Apache system, we discovered a great deal about the inner performance dependencies of the system, while simultaneously revealing performance bottlenecks and poor design decisions (from a performance per-spective); our conclusions about the Pentium Pro's memory system and Apache's subopti-mal synchronization policy highlight the types of conclusions that can be garnered from the detail of a typical performance hierarchy.

Finally, despite its obvious success in the cases we examined, the performance decom-position approach is not yet developed to its full potential. For example, the distillation methodology described in Chapter 4 is still quite rough; with more careful analysis of the correlation between OS abstractions and *hbench-OS* primitives, the accuracy and absolute predictive power of the generated metrics could certainly be enhanced. Similarly, a further broadening of the *hbench-OS* tests could improve the accuracy of the characterization and thus also enhance the predictive power of the generated metrics.

Even in its current form, however, the performance decomposition methodology pre-sented in this thesis is a powerful tool for understanding the performance of modern com-puter systems; with further refinement, it promises to be even more effective in stripping away complexity to reveal the true nature of modern system performance.

References

[1]   Anderson, T., Levy, H., Bershad, B., and Lazowska, E. "The Interaction of Architecture and Operating System Design." *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1991: 108–120.

[2]   "Apache HTTP Server Project." http://www.apache.org.

[3]   Chen, B., and 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., Mazières, D., Dias, A., Seltzer, M., and 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.* 1st ed. Sebastopol, CA: O'Reilly and Associates, 1994.

[6]   "Etch Overview." http://www.cs.washington.edu/homes/bershad/etch/index.html.

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

[8]   Graham, S., Kessler, P., Muckiest, M. "gprof: a Call Graph Execution Profiler." *ACM SIGPLAN Notices* 17.6 (1982): 120–126.

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

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

[11]   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.

[12]   Maynard, A., Donnelly, C., and 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.

[13]   McKusick, M. "Using gprof to Tune the 4.2BSD Kernel (May 21, 1984)." Distributed with 4.4BSD UNIX, 1993.

[14]   McVoy, L., and Staelin, C. "lmbench: Portable Tools for Performance Analysis." *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, January 1996: 279–294.

[15]   "The NetBSD Project." http://www.netbsd.org.

[16]   Nishioka, S., Kawaguchi, A., and Motoda, H. "Process-labeled Kernel Profiling: A New Facility to Profile System Activities." *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, January 1996: 295–306.

[17]   Ousterhout, J. "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.

[18]   Patterson, D., and Hennessy, J. *Computer Architecture A Quantitative Approach.* 2nd ed. San Francisco: Morgan Kaufmann, 1996.

[19]   Rosenblum, M., Bugnion, E., Herrod, S., Witchel, E., and 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.

[20]   Rosenblum, M., Bugnion, E., Devine, S., and Herrod, S. "Using the SimOS Machine Simulator to Study Complex Computer Systems." *to appear in ACM Trans. on Modeling and Computer Simulation Special Issue on Computer Simulation*, 1997.

[21]   Saavedra-Barrera, R. H., Smith, A. J., and Miya, E. "Machine Characterization Based on an Abstract High-Level Language Machine" *IEEE Transactions on Computers* 38 (1989): 1659–1679.

[22]   Saavedra, R. H., Smith, A. J., "Analysis of Benchmark Characteristics and Benchmark Performance Prediction," *ACM Transactions on Computer Systems* 14 (1996): 344–384.

[23]   Singh, J., Weber, W., and Gupta, A. "SPLASH: Stanford Parallel Applications for Shared-Memory." *Computer Architecture News* 20.1 (1993): 5–44.

[24]   "SPEC CPU95." http://www.specbench.org/osg/cpu95.

Acknowledgments

I wish above all to thank my advisor, Margo Seltzer, without whose insight, enthusiasm, and support this thesis would never have been written. I also want to thank H. T. Kung for encouraging me to look beyond basic benchmarking; his insightful comments motivated much of the characterization and distillation work of Chapter 4. Many thanks go as well to all those who provided technical advice and moral support throughout the entire process of creating this thesis, especially the Vino group, my roommates Joshua Greene and Michael Kim, my parents, my fellow thesis writers Steve Chien, Steve Manley, and Lori Park, Noemí Flores, Hua Tang, and many others. Finally, I thank my readers, H. T. Kung and Michael Smith, for willingly agreeing to take the time to read this thesis.