Making the “Box” Transparent: System Call Performance as a First-class Result [*]

Yaoping Ruan and Vivek Pai
Department of Computer Science
Princeton University
{yruan,vivek}@cs.princeton.edu

Abstract

For applications that make heavy use of the operating system, the ability of designers to understand system call performance behavior may be essential to achieving high performance. Conventional approaches to performance analysis, such as monitoring tools and profilers, collect and present their information off-line or via out-of-band channels. We believe that making this information *first-class* and exposing it to running applications via *in-band* channels on a *per-call* basis presents opportunities for analysis and performance tuning not available via other mechanisms. Furthermore, our approach provides direct feedback to applications on time spent in the kernel, resource contention, and time spent blocked, allowing them to immediately observe how the application and workload affect kernel behavior. Not only does this approach provide greater *transparency* into the workings of the kernel, but it also allows applications to control how performance information is collected, filtered, and correlated with application-level events.

To demonstrate the power of this approach, we show that our implementation, DeBox, obtains precise information about OS behavior at low cost, and that it can be used in debugging/tuning application performance on complex workloads. In particular, we focus on the industry-standard SpecWeb99 benchmark running on the Flash Web Server. Using DeBox, we are able to diagnose a series of problematic interactions between the server and the operating system. Addressing these issues as well as other optimization opportunities generates an overall factor of four improvement in our SpecWeb99 score and throughput gains on other benchmarks. Equally importantly, our measurements suggest that parallelism stemming from programmer convenience has a sharply negative impact on latency. We show how our optimizations reduce this impact, improving latency from a factor of 4 to 47 under different conditions.

1 Introduction

Operating system performance continues to be an active area of research, especially as demanding applications test OS scalability and performance limits. The kernel/user boundary becomes critically important as these applications spend a significant fraction, often a majority, of their time executing system calls. In the past, developers could expect to put data-sharing services, such as NFS, into the kernel to avoid the limitations stemming from running in user space. However, with the rapid rate of developments in HTTP servers, Web proxy servers, peer-to-peer systems, and other networked systems, using kernel integration to avoid performance problems becomes unrealistic. As a result, examining the interaction between operating systems and user processes remains a useful area of investigation.

Much of the earlier work focusing on the kernel/user interface centered around the developing new system calls more closely tailored to the needs of particular applications. In particular, zero-copy I/O [18, 35] and scalable event delivery [8, 9, 26] are examples of techniques that have been adopted in mainstream operating systems, via calls such as `sendfile()`, `transmitfile()`, `kevent()`, and `epoll()`, to address performance issues for servers. Other approaches, such as allowing processes to declare their intentions to the OS [36], have also been proposed and implemented. Some system calls, such as `madvise()`, provide some hints to the OS, but with operating systems free to ignore such requests or restrict them to mapped files, programs cannot rely on their behavior.

Some recent research uses the reverse approach, where applications determine how the “black box” OS is likely to behave and then adapt accordingly. For example, the Flash Web Server [34] uses the `mincore()` system call to determine memory residency of pages, and combines this information with some heuristics to avoid blocking. The “gray box” approach [7, 15] tries to infer memory residency by observing page faults and correlating them with known replacement algorithms. In both systems, memory-resident files are treated differently than others, improving performance and/or latency. These approaches depend on the quality of the information they can obtain from the operating system and the accuracy of their heuristics. As workload complexity increases, we believe that such inferences will become harder to make.

To remedy these problems, we propose a much more direct approach to making the OS transparent: make system call performance information a first-class result, and return it in-band. In practice, what this entails is having each system call fill a “performance result” structure, providing information about what occurred in processing the

[*]This work has been partially supported by an NSF CAREER award

call. While it is much larger and more detailed than the `errno` global variable, they are conceptually similar. Simple monitoring at the system call boundary, the scheduler, page fault handlers, and function entry/exit is sufficient to provide detailed information about the inner working of the operating system. This approach not only eliminates guesswork about what happened during call processing, but it gives the application control over how this information is collected, filtered, and analyzed, providing more customizable and narrowly-targeted performance debugging than is available in existing tools, yielding poorer results.

We evaluate the flexibility and performance of our implementation, DeBox, running on the FreeBSD operating system. DeBox allows us to determine where applications spend their time inside the kernel, what causes them to lose performance, what resources are being contended, and how the kernel behavior changes with the workload. The flexibility of DeBox allows us to measure very specific information, such as the kernel CPU consumption caused by a single call site in a program.

Our throughput experiments focus on analyzing and optimizing the performance of the Flash Web Server on the industry-standard SpecWeb99 benchmark [44]. Using DeBox, we are able to diagnose a series of problematic interactions between the server and the operating system on this benchmark. Addressing these issues as well as other optimization opportunities generates an overall factor of four improvement in our SpecWeb99 score and throughput gains on other benchmarks.

Equally importantly, our measurements suggest that parallelism stemming from programmer convenience, which we term *excess parallelism*, has a sharply negative impact on latency. We find that the sources of latency are not specific to Flash or SpecWeb99 – even using other servers with other workloads, we are able to demonstrate response latency increasing with excess parallelism and trace their sources. We show how our optimizations reduce this impact, improving latency from a factor of 4 to 47 under different conditions.

The rest of this paper is organized as follows. Section 2 provides some background on performance analysis and monitoring tools. In Section 3 we discuss some motivating examples where existing tools do not suffice. The detailed DeBox design and implementation are described in Section 4. We conduct our case study of how we use DeBox to analyze and optimize the Flash Web Server in Section 5. We summarize the results of the case study and modifications in Section 6 and conduct some experiments on latency in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2 Background

By making performance information a first-class result of the system call, DeBox allows applications to gain some of the information provided by other monitoring/profiling tools. More importantly, it allows applications to tailor the information provided to them in ways that would be difficult with out-of-band approaches or where information is aggregated before being presented. We describe some of the different types of tools currently in use, and how DeBox relates to these approaches. Note that replacing all of these tools is an explicit non-goal of DeBox, nor do we believe that such a goal is even feasible.

- **Function-based profilers** – One of the most common and effective means of detecting performance hotspots in programs and kernels is the use of function-based profilers, such as `prof`, `gprof` [21, 22], and their variants. These tools use compiler assistance to add bookkeeping information (count and time) to the entry and exit points of functions. This information is gathered while running and analyzed offline to reveal function call counts and CPU usage, often along paths in the call graph. This approach often suffers from high overhead, especially when function call times are small.
- **Coverage-based profilers** – These profilers divide the program of interest into regions and use a clock interrupt to periodically sample the location of the program counter. Like function-based profilers, data gathering is done online while analysis is performed offline. Tools such as `profil()`, `kernbb`, and `tcov` can then use this information to show what parts of the program are most likely to consume CPU time. Coverage-only approaches may miss infrequently-called functions entirely and may not be able to show call-graph behavior. Coverage information combined with compiler analysis can be used to show usage on a basic-block basis.
- **Hardware-assisted profilers** – These profilers are similar to coverage-based profilers, but use special features of the microprocessor (event counters, timers, programmable interrupts) to obtain higher-precision information at lower cost. The other major difference is that these profilers, such as DCPI [4], Morph [48], VTune [23], Oprofile [33], and PP[3] tend to be whole-system profilers, capturing activity across all processes and the operating system. DCPI, in particular, is designed to have low enough overhead to run continuously in the background.

With respect to profilers, DeBox is logically closest to kernel `gprof`, though it provides more than just timing information. Call graphs can be constructed from its call tracing output, and with the data compression/storage performed in user space, overhead is moved from the kernel to the process. Coverage differs, however, since DeBox only measures functions directly used during the system call. As a result, interrupt handlers and related functions, such as the bottom half of the network stack, are not included.

- **System activity monitors** – Tools such as `top`, `vmstat`, `netstat`, `iostat`, and `systat` are commonly used by system administrators to monitor a running system or by users trying to determine a first-order cause for system slowdowns. The level of precision for the various tools varies greatly, with `top` showing per-process information on CPU usage, memory consumption, ownership, and running time, to `vmstat` showing only summary information on memory usage, fault rates, disk activity, and CPU usage.

- **Trace tools** – Trace tools provide a means of observing the system call behavior of processes without requiring access to process source or modifying it in any way. These tools, such as `truss`, PCT [11] and `strace` [2], are able to show various levels of details of system calls, such as parameter values, return values, and timing/count information. More recent tools, such as `ktrace` and the Linux Trace Toolkit [47], also provide insight into some of the kernel state that changes as a result of the system calls. These tools are intended for observing another process, and as a result, produce out-of-band measurements, often requiring post-processing to generate usable output.
- **Timing calls** – One of the simplest approaches usable by programmers is to manually record the start and end times of certain events (e.g., using `gettimeofday()` or similar calls), and to try inferring information based on the difference. The `getrusage()` call additionally adds some other information beyond timings (context switches, faults, messages and I/O counts) and can similarly used.

DeBox compares favorably with a combination of the timing calls and the trace tools in the sense that timing information is presented in-band, but the level of detail is comparable to what is provided by the trace tools. Our current prototype does not include the level of detail provided by the Linux Trace Toolkit, but the basic structure is amenable to expansion, and we are investigating the utility of that level of information.

- **Microbenchmarks** – Popular tools for measuring best-case times or the actual cost of certain operations (cache misses, context switches, etc.) can be obtained from microbenchmarks such as lmbench [28] and hbench:OS [13]. Common usage for these tools is to compare different operating systems, different hardware platforms, or possible optimizations.
- **Latency tools** – Recent work on attempting to find the source of latency on desktop systems not designed for real-time work have yielded insight and some tools. The Intel Real-Time Performance Analyzer [37] helps automate the process of pinpointing latency. The work of Cota-Robles and Held [16] and Jones and Regehr [24] demonstrate the benefits of successive measurement and searching.
- **Instrumentation** – Dynamic instrumentation tools provide mechanisms to instrument running systems (processes or the kernel) under user control, and to obtain precise kernel information. Examples include DynInst [14], KernInst [45], ParaDyn [29], Etch [39], and ATOM [42]. The appeal of this approach versus standard profilers is the flexibility (arbitrary code can be inserted) and the cost (no overhead until use). Information is presented out-of-band.

Since DeBox measures the performance of calls in their natural usage, it resembles the instrumentation tools. DeBox gains some flexibility by presenting this data to the application, which can filter it on-line. The main difference between DeBox and kernel instrumentation is that we provide a standard set of measurements to any process, rather than providing more detail to processes allowed to modify the kernel.

3 Motivation

DeBox is designed to bridge the divide in performance analysis across the kernel/user boundary by exposing kernel performance behavior to user processes. The primary motivation behind DeBox is to enable performance debugging and analysis of server-style applications on demanding workloads. In these environments, performance problems can occur on either side of the boundary, and limiting analysis to only one side potentially eliminates useful information. Even though some servers may spend most of their time in the kernel, the ultimate cause may be activities under process control. As a result, applications may be able to modify their own behavior to avoid bottlenecks. Additionally, by making performance information first-class, we believe that DeBox provides opportunities not afforded by out-of-band or off-line approaches. Some examples are provided below.

User-level timing approaches are not sufficient. In complex workloads, user-level timing approaches can not be used to reliably infer the presence of unusual activity in the kernel. Complex workloads can arise when many processes are competing for the CPU, as in multiprocessor servers, or in event-driven servers when dynamic content is run in separate processes. Figure 1 shows user-level timing measurement of the `sendfile()` system call in an event-driven server. This server uses nonblocking sockets and invokes sendfile only when it believes the data to be sent is present in main memory. As a result, the presence of peaks on this graph is a cause for concern, because they may indicate that the server is blocking. In reality, though, the user-level timing functions around sendfile present a small window of opportunity for the scheduler to be invoked.

Using DeBox, timing measurement is integrated into the system call process, and does not suffer from measurement errors caused by scheduling between the system call and the timing. The DeBox-derived measurements of the same call are shown in Figure 2, and do not indicate such sharp peaks. Summary data for `sendfile` and `accept` are shown in Table 1. Since DeBox also monitors the scheduler, if a system call blocks and another process is run, the timing information reflects both the wall-clock time for the system call, in addition to the actual time used. One area of weakness in many systems is the lack of proper accounting for interrupts, and DeBox does not attempt to remedy this shortfall on its own. However, other approaches, such as Lazy Receiver Processing [17] or Deferred Procedure

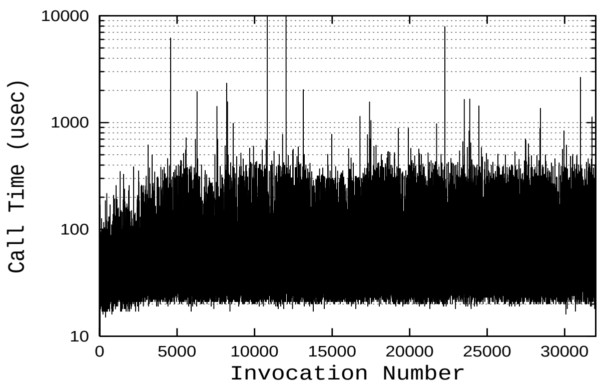


Figure 1: **User-space timing of the `sendfile` call on a server running the SpecWeb99 benchmark.** Note the sharp peaks, which may indicate anomalous behavior in the kernel.

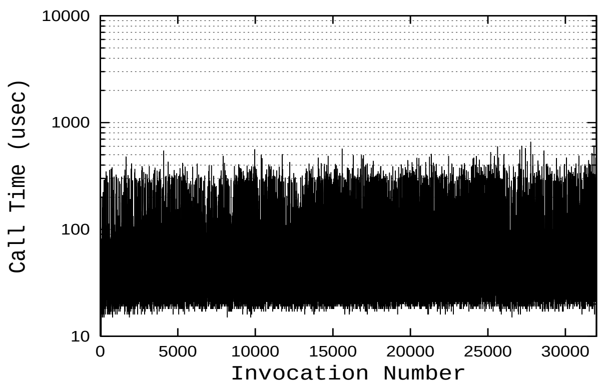


Figure 2: **The same system call measured using DeBox shows much less variation in behavior.**

Calls [24], can be used to mitigate this effect.

	`accept()`		`sendfile()`	
	User	DeBox	User	DeBox
Min	5.0	5.0	8.0	6.0
Median	10.0	6.0	60.0	53.0
Mean	14.8	10.5	86.6	77.5
Max	5216.0	174.0	12952.0	998.0

Table 1: **Execution time (in usec) of two system calls measured in user application and DeBox.** Note the large difference in maximums stemming from the measuring technique.

Standard kernel profiling activation is binary. In general use, standard kernel profiling monitors all of the kernel, and must be activated by the superuser. Due to the high cost of call-graph profiling, the restriction on activation is sensible. However, this restriction prevents ordinary application developers from using such tools. The other drawback to the overhead is that it may distort the bottlenecks in the system. If call-graph profilers generate a factor of two slowdown in the system (not uncommon), then the CPU may become the bottleneck resource when the profiler is running, even though it may not be the actual bottleneck. In these situations, profiling can cause overflowing queues, delayed responses, and all of the other effects of overload.

By making performance information a result of system calls, the overhead normally associated with kernel call-graph profiling is reduced in two ways. The first is that applications may individually opt to profile their interaction with the kernel, eliminating the binary nature of kernel profiling. Not only does this approach reduce the amount of profiling done, but one process desiring profiling does not affect the behavior of others on the system. The second aspect is that the work of collecting and storing the profiling information is shifted to the application, instead of being the responsibility of the system kernel. In this way, applications affect only their own share of system resources if they perform the processing normally associated with call-graph profilers.

Off-line reporting discards useful information. Servers handling thousands of requests per second are not uncommon, and even with only a few system calls per request, profilers may have to record tens of thousands of system calls per second. Obviously, capturing the full detail of each call is infeasible, since the recording process would require more effort than actually performing the system call itself. As a result, sampling approaches or information aggregation are necessary in profilers and monitoring tools. However, the information of interest may be lost in the aggregation/filtering process.

When applications receive performance information along with each system call, they can provide the filtering and aggregation that they need in order to accomplish their goals. Applications may consider different things "interesting": calls that take longer than usual, calls that block, or the change in call time over the course of the program. In cases such as these, applications may opt to throw away most of the data, and record some of it in much greater detail as needed. Applications may also be able to correlate information about system calls with the underlying action that is causing them. This concept is similar to how `gprof` separates profile information along different edges of a call graph. However, applications may use information not available at compile-time in order to make their classification decisions. The key point in all of these cases is that using DeBox, the applications maintain control over information filtering/aggregation.

4 Design & Implementation

This section describes our DeBox prototype implementation in FreeBSD and measures its overhead. We first describe the user-visible portion of DeBox, and then the kernel modifications. We also measure the overhead of having DeBox support in the kernel, as well as the cost for using it. Examples of how to fully use DeBox and what kinds of information it provides are deferred to the case study in Section 5.

```
typedef struct PerSleepInfo {
  int numSleeps;              /* # sleeps for the same reason */
  struct timeval blockedTime; /* how long the process is blocked */
  char wmesg[8];              /* reason for sleep (resource label) */
  char blockingFile[32];      /* file name causing the sleep */
  int blockingLine;           /* line number causing the sleep */
  int numWaitersEntry;        /* # of contenders at sleep */
  int numWaitersExit;         /* # of contenders at wake-up */
} PerSleepInfo;

typedef struct CallTrace {
  unsigned long callSite;     /* address of the caller */
  int deltaTime;              /* elapsed time in timer or CPU counter */
} CallTrace;

typedef struct DeBoxInfo {
  int syscallNum;             /* which system call */
  union CallTime {
    struct timeval callTimeval;
    long callCycles;          /* wall-clock time of entire call */
  } CallTime;
  int numPGFaults;            /* # page faults */
  int numPerSleepInfo;        /* # of filled PerSleepInfo elements */
  int traceDepth;             /* # functions called in this system call */
  struct PerSleepInfo psi[5]; /* sleeping info for this call */
  struct CallTrace ct[200];   /* call trace info for this call */
} DeBoxInfo;

int DeBoxControl(DeBoxInfo *resultBuf, int maxSleeps, int maxTrace);
```

Figure 3: **DeBox data structures and function prototype**

4.1 User-Visible Portion

The programmer-visible interface to DeBox is intentionally simple, since it consists of some monitoring data structures and a new system call to enable/disable data gathering. Figure 3 shows the data structure that handles the DeBox information, DeBoxInfo. It serves as the "performance information" counterpart to other system call results like `errno`. Programs wishing to use DeBox need to perform two actions: declare one or more of these structures as global variables, and call DeBoxControl to inform the operating system of the amount of per-call performance information it wishes to obtain.

At first glance, the DeBoxInfo structure appears to be very large, which would normally be an issue since its size could affect system call performance. This structure size is not a significant concern, since the entire structure is rarely copied, and the process can specify limits on how much of it is used. Most of the space is consumed by two arrays, and they are not expected to be fully used in practice. The PerSleepInfo array contains information about each of the times the system call blocks (sleeps) in the course of processing. The CallTrace array provides the history of what functions were called and how much time was spent in each. Both arrays are generously sized, and we do not expect many calls to fully utilize either one.

DeBoxControl can be called multiple times over the course of a process execution for a variety of reasons. Programmers may wish to have several DeBoxInfo structures and use different structures for different purposes. They can also vary the number of PerSleepInfo and CallTrace items recorded for each call, to vary the level of detail generated. Finally, they can specify a NULL value for resultBuf, which deactivates DeBox monitoring for the process.

4.2 In-Kernel Implementation

The kernel portion of DeBox consists largely of performing the necessary bookkeeping to gather the data in the DeBoxInfo structure. The points of interest are the system call entry/exit, the scheduler sleep and wakeup routines, and the function entry/exit for all functions reachable from a system call.

Since DeBox returns performance information when each system call finishes, the system call entry/exit code is modified to detect if a process is using DeBox. Once a process calls DeBoxControl and specifies how much of the arrays to use, the kernel stores this information and allocates a kernel-space DeBoxInfo reachable from the process control block. This copy is used to record information while the system call executes, consolidating the data gathering that would otherwise require a large number of small kernel/user copies. Prior to system call return, the requested

```
DeBoxInfo:
        4, /* system call # */
  3591064, /* call time, microsecs */
      989, /* # of page faults */
        2, /* # of PerSleepInfo used */
        0, /* # of CallTrace used (disabled) */

PerSleepInfo[0]:                                          PerSleepInfo[1]:
          1270  /* # occurrences */                                    325
        723903  /* time blocked, microsecs */                      2710256
         biowr  /* resource label */                                spread
 kern/vfs_bio.c  /* file where blocked */      miscfs/specfs/spec_vnops.c
          2727  /* line where blocked */                               729
             1  /* # processes on entry */                               1
             0  /* # processes on exit */                                0
```

Figure 4: **Sample DeBox output showing the system call performance of copying a 10MB mapped file**

information is copied back to user space.

At system call entry, all non-array fields of the process's DeBoxInfo are cleared. Arrays do not need to be explicitly cleared since the counters indicating their utilization have been cleared. Call number and start time are stored in the entry. We measure time using the CPU cycle counter available on our hardware, but we could also use timer interrupts or other facilities provided by the hardware.

Page faults that occur during the system call are counted by modifying the page fault handler to check for DeBox activation. We currently do not provide more detailed information on where faults occur, largely because we have not observed a real need for this information. However, since the DeBoxInfo structure can contain other arrays, more detailed page fault information can be added in the future if desired.

The most detailed accounting in DeBoxInfo revolves around the "sleeps," when the system call blocks waiting on some resource. When this occurs in FreeBSD, the system call invokes the `tsleep()` function, which passes control to the scheduler. When the resource becomes available, the `wakeup()` function is invoked and the affected processes are unblocked. In FreeBSD, kernel routines invoking the `tsleep()` mechanism provide a human-readable label for use in utilities like `top`. We define a new macro for `tsleep()` in the kernel header files that permits us to intercept any sleep points. When this occurs, we record in a PerSleepInfo element where the sleep occurred (blockingFile/blockingLine), what time it started, what resource label was involved (wmesg), and the number of other processes waiting on the same resource (numWaitersEntry). Similarly, we modify the `wakeup()` routine to provide numWaitersExit and calculate how much time was spent blocked. If the system call sleeps more than once at the same location, that information is aggregated into a single PerSleepInfo entry.

The process of tracing which kernel functions are called during system call processing is slightly more involved, largely to minimize overhead. Conceptually, all that has to occur is that every function entry and exit point has to record that it was executed and when it started/finished, similar to what call graph profilers use. The gcc compiler allows entry/exit functions to be specified via the "instrument functions" option, but these are invoked by explicit function calls. As a result, function call overhead increases by roughly a factor of three. Our current solution involves manually inserting entry/exit macros into reachable functions and recording the function address and timings into the CallTrace array. Automating this modification process should be possible in the future, and we are investigating using the `mcount()` kernel function used for kernel profiling.

To get a sense of what kind of information is provided in DeBox, we show sample output in Figure 4. We begin by memory-mapping a 10MB file, and then using the `write()` system call to copy its contents to another file. The main DeBoxInfo structure shows that system call 4 (`write()`) was invoked, and it ran for about 3.6 seconds of wall-clock time. It incurred 989 page faults, and blocked in two unique places in the kernel. The first element of the PerSleepInfo array shows that it blocked 1270 times at line 2727 in vfs_bio.c on "biowr", which is the block I/O write routine. The second location was line 729 of spec_vnops.c, which caused 325 blocks at "spread", a read of a special file. The writes blocked for roughly 0.7 seconds, and the reads for 2.7 seconds.

4.3 Overhead

For DeBox to be attractive, it should generate low kernel overhead, especially in the common case. To quantify this overhead, we compare an unmodified kernel, a kernel with DeBox support, and the modified kernel with DeBox activated. We show these measurements in Table 2. The first column indicates the various system calls – `getpid()`, `gettimeofday()`, and `pread()` with various sizes. The second column indicates the time required for these calls on an unmodified system. The remaining columns indicate the additional overhead for various DeBox features on a modified system.

call name or read size	base time	basic off	basic on	trace off	trace on
getpid	0.46	+0.00	+0.50	+0.03	+1.45
gettimeofday	5.07	+0.00	+0.43	+0.03	+1.52
pread 128B	3.27	+0.02	+0.56	+0.21	+2.03
256 bytes	3.83	+0.00	+0.59	+0.26	+2.02
512 bytes	4.70	+0.00	+0.69	+0.28	+2.02
1024 bytes	6.74	+0.00	+0.68	+0.27	+2.02
2048 bytes	10.58	+0.03	+0.68	+0.26	+2.01
4096 bytes	18.43	+0.03	+0.74	+0.29	+2.16

Table 2: **DeBox microbenchmark overheads:** Base time is the execution time on an unmodified system. All times are in microseconds

We find that the largest source of performance loss is call history tracing, so we separate its measurement. The "basic off" column indicates the overhead introduced with a modified kernel supporting DeBox without call tracing. The performance impact is virtually unnoticeable. The "basic on" column show the impact of activating DeBox without call tracing. We use the CPU cycle counter, since accessing the hardware clock on our system requires 5 microseconds. This overhead is the reason why gettimeofday has a comparable running time to a 512 byte read.

From these numbers, we can see that the cost to support most DeBox features is minimal, and the cost of using the measurement infrastructure is tolerable. Since these costs are borne only by the applications that choose to enable DeBox, the overhead is tolerable. The cost of supporting call tracing, shown in the "trace off" column, where every function entry and exit point is affected, is higher, averaging approximately 5% of the system call time. This overhead is higher than ideal, and may not be desirable to have continuously enabled. However, our implementation is admittedly crude, and better compiler support could better integrate it with the function prologue/epilogue code. We expect that we can reduce this overhead, along with the overhead of using the call tracing, with optimization.

The overhead of microbenchmarks do not indicate what kinds of slowdowns may be typically observed. To give some insight into these costs, Table 3 shows some macrobenchmarks on an unmodified system, one with only "basic" DeBox activated, and one with complete DeBox support. The first two columns are times for archiving and compressing files of different sizes. The last column is for building the kernel. The overheads of DeBox support range from less than 1 percent to roughly 3 percent in the kernel build. We expect that many environments will tolerate this overhead in exchange for the flexibility provided by DeBox.

	tar—gz a directory with		make
	1MB file	10MB file	kernel
base time	275.61 msec	3078.50 msec	236.96 sec
basic on	+0.97 msec	+22.73 msec	+1.74 sec
full support	+1.03 msec	+44.58 msec	+7.49 sec

Table 3: **DeBox macrobenchmark overheads**

5 Case Study

In this section, we show a case study of using DeBox to analyze and optimize the behavior of the Flash Web Server running on the FreeBSD operating system. We discover a series of problematic interactions, trace their causes, and find appropriate solutions to avoid them or fix them. In the process, we gain insights into the causes of performance problems and how seemingly simple solutions, such as throwing more resources at the problem, may exacerbate the problem. Our optimizations generate an overall factor of four improvement in our result on the SpecWeb99 benchmark and also lead to a sharp decrease in latency.

5.1 Experimental Setup & Workload

We first describe our experimental setup and the relevant software components of the system. All of our experiments are performed on a uniprocessor server running FreeBSD 4.6, with a 933MHz Pentium III, 1GB of memory, one 5400 RPM Maxtor Diamond IDE disk, one Promise Ultra DMA 66 controller, and a single Netgear GA621 gigabit ethernet network adapter. The clients consist of ten Pentium II machines running at 300 MHz connected to a switch using Fast Ethernet. All machines are configured to use the default (1500 byte) MTU as required by the SpecWeb99 benchmark.

Our main application is the event-driven Flash Web Server, although we also perform some tests on the widely-used multi-process Apache [6] server. The Flash Web Server consists of a main process and a number of helper processes. The main process multiplexes all client connections, is intended to be nonblocking, and is expected to serve all requests only from memory. The helpers load disk data and metadata into memory to allow the main process to avoid blocking on disk. The number of main processes in the system is generally equal to the number of physical processors, while the the number of helper processes is tied to the number of disks, and is dynamically adjusted based on load. In previous tests, the Flash Web Server has been shown to compare favorably to high-performance commercial Web servers [34]. We run with logging disabled.

We focus on the SpecWeb99 benchmark, an industry-standard workload that is designed to test the overall scalability of Web servers under realistic conditions. It is designed by SPEC, the developers of the widely-used SpecINT and SpecFP workloads [43], and its parameters are derived from observations of workloads at production Web sites. Although not common in academia, it is the *de facto* standard in industry [31], with over 150 published results, and is different from most other Web server benchmarks in its complexity and requirements. It measures the overall scalability of a system by reporting the number of simultaneous connections the server is able to handle while meeting a specified quality of service. The sizes of the data set and working set increase with the number of simultaneous connections, and quickly exceed the physical memory of commodity systems. 70% of the requests are for static content, with the other 30% for dynamic content, including a mix of HTTP GET and POST requests. 0.15% of the requests require the use of a CGI process that must be spawned separately for each request.

5.2 Initial experiments

Our first run of SpecWeb99 on the publically-available version of the Flash Web Server yields a SpecWeb99 result of roughly 200 simultaneous connections, much lower than the published score of 575 achieved by comparable hardware. At 200 simultaneous connections, the data set size is roughly 750MB, which is smaller than the amount of physical memory in the machine. Not surprisingly, the workload is CPU-bound, and a quick examination shows that the `mincore()` system call is consuming more resources than any other call site in Flash.

The underlying problem is the use of linked lists in the FreeBSD virtual memory subsystem for handling virtual memory objects. The heavy use of memory-mapped files in Flash generates large numbers of memory objects, and a linear walk utilized by `mincore()` generates significant overhead. We apply a patch from Alan Cox of Rice University that replaces the linked list with a splay tree, and this brings `mincore()` in line with other calls. Our SpecWeb99 score rises to roughly 320, a 60% improvement. At this point, the working set has increased to 1.1GB, slightly exceeding our physical memory.

5.3 Modern Interfaces

Once the `mincore()` problem is addressed, we find that the two most CPU-intensive system calls are `select()` and `writev()`. The former is used to determine which file descriptors are ready for service, while the latter is used to send data back to the client. Since the development of Flash, FreeBSD has incorporated a zero-copy I/O system call, `sendfile()`, and a scalable event delivery mechanism, `kevent()`. With memory-mapped files, Flash generally closes the associated descriptor, reducing the impact on `select()`. However, using `sendfile()` requires that file descriptors be kept open, greatly increasing the number of file descriptors in use by Flash. To mitigate this impact, we implement support for `sendfile()` concurrently with support for `kevent()`.

CPU utilization drops after these changes are introduced, but to our surprise, so does performance. Using `writev()` instead of `sendfile()` seems to make little difference in performance, an observation we note for later investigation. We find the CPU has idle time, but when we attempt to increase the offered load, we find that Flash is not able to meet the quality-of-service requirements of SpecWeb99. One obvious cause for this kind of situation is that the server is blocking, so additional load can not allow the server to take advantage of the available CPU.

biord/166	inode/127	getblk/1	sfpbsy/1
open/162	readlink/84	close/1	sendfile/1
read/3	open/28		
unlink/1	read/9		
	stat/6		

Table 4: **Summarized DeBox output showing blocking counts:** The layout is organized by resource label and system call name. For example, of the 127 times this test blocked with the “inode” label, 28 were from the `open()` system call

Using the PerSleepInfo data, we record every system call invocation from the main Flash process that blocks inside the kernel. The main process is designed to be non-blocking, so any blocking system call is of interest. The results of this data gathering are shown in Table 4, where each column header shows the resource label (wmesg) causing the blocking, followed by the total number of times blocked at that label. The elements in the column are the system calls that block on that resource, and the number of invocations involved. As evidenced by the calls involved, the “biord” (block I/O read) and “inode” (vnode lock) labels are both involved in opening and retrieving files from disk, which is not surprising since our data set exceeds the physical memory of the machine.

5.4 Revisiting Flash Helpers

For portability, the main process in Flash only uses the helpers to demand-fetch disk data and metadata into the OS caches, but does not otherwise use their results. One set of helpers is used to resolve URLs to files on disk, and are known as the name conversion helpers. Once the helpers have completed loading data, the main process repeats the operation immediately, assuming that the recently loaded information will prevent it from blocking. Observing the timings of system call activity, we find that when the main process blocks, the helper processes are operating on similarly-named files as the main process.

# of helpers	1	5	10	15
Blocking count	114	295	339	394
% Conforming	40.9%	95.1%	96.9%	89.5%

Table 5: **Parallelism benefits and self-interference:** The conformance measurement indicates how many requests meet SpecWeb99’s quality-of-service requirement.

Guided by this information, we determine that the interference between the main process and the helpers occurs when they access files which share path components. To test our hypothesis, we try increasing the number of helper processes and observe its effect on the SpecWeb99 results, as shown in Table 5. We observe that too few helpers is insufficient to fully utilize the disk, and increasing their number initially helps performance. However, note that the number of blocks from self-interference increases, eventually decreasing performance. This self-interference may affect other systems that try to use parallelism to increase performance [46]. We solve this problem by having the helper processes return open file descriptors using `sendmsg()`, eliminating duplication of work in the main process.

We find that this change alone solves most of the filesystem-related blocking. However, one `open()` call in Flash still shows periodic blocking at the label "biord" (reading a disk block), but only after the server has been running for some time. To determine what application path causes this behavior, we have the process call `abort()` when `open()` sleeps, such that we can examine the user stack trace and data structures.

This problem uncovers a subtle performance bug in Flash induced by mapped-file cache replacement. Flash has two independent caches – one for URL-to-filename translations (name cache), and another for memory-mmaped regions (data cache). For this workload, the name cache does not suffer from capacity misses, while the data cache may evict the least recently used entries. Under heavy load, a name cache hit and a data cache capacity miss causes Flash to erroneously believe that it had just recently performed the name translation. When Flash calls `open()` to access the file, the metadata associated with the name conversion is missing, causing blocking. We solve this problem by allowing the second set of helpers, the read helpers, to return file descriptors if the main process does not already have them open.

The final source of metadata-related blocking is diagnosed by using the kernel call tracing facilities of DeBox and determining what paths get executed in the miss cases. We discover that in FreeBSD, two parameters control the metadata cache policy, "vmiodirenable" and "nameileafonly". The former determines if directory metadata caching can use the block cache, and the second determines if non-leaf metadata cache entries can be evicted. We enable both options, and the remaining metadata-related blocking disappears. With these change, we are able to handle 390 simultaneous connections from SpecWeb99, with a data set size of 1.3GB.

5.5 Process Creation Overhead

With all blocking eliminated and with a much higher request rate, we return to the issue of system call CPU consumption and find that the largest call times are for the `fork()` system call. These calls stem from the SpecWeb99 workload requirement that 0.15% of the requests be handled by forking off new processes. Among the system calls, we discover that `fork()` takes as long as 130ms, while most calls finish in 1 ms. Using DeBox's ability to measure the per-call time, we record the per-call time as function of call number, to generate Figure 5. We observe that `fork()` time increases as the program runs, starting as low as 300 microseconds.

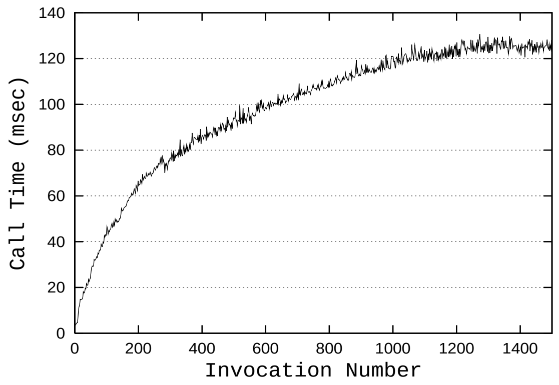


Figure 5: **Call time of `fork()` as a function of invocation number**

Call tracing measurements indicate that copying mapped regions and file descriptors during `fork()` is consuming most of the time. We confirm this observation by varying the sizes of the caches in Flash and seeing their impact on `fork()` times. Rather than try to address this by changing the implementation of `fork()`, we opt to slightly modify the Flash architecture. We introduce a new helper process that is responsible for the creation of the CGI processes. Since this new process does not map files or cache open files, its `fork()` time is not affected by the main process size. This change yields a 10% improvement, to 440 simultaneous connections and a 1.5GB data set size.

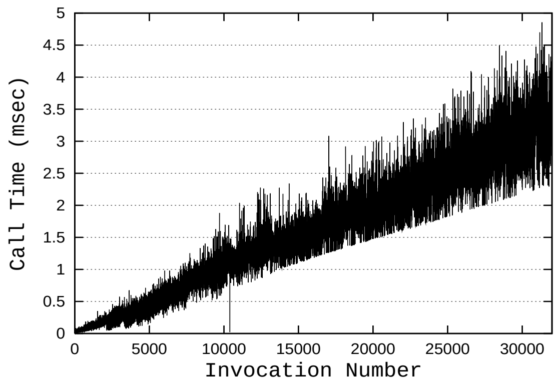


Figure 6: **Call time of `mmap()` as a function of invocation number**

5.6 Memory residency overhead

At this point, even though the data set size exceeds physical memory by over 50%, the system bottleneck is the CPU, largely due to the amount of overhead involved in memory residency checking. Though our modified Flash uses `sendfile()` and does not examine file content, the use of `mincore()` to determine memory residency requires that files be memory-mapped. The cumulative overhead of memory-map operations is the largest consumer of CPU time. As can be seen in Figure 6, the per-call overhead of `mmap()` is significant and increases as the server runs. The cost increase is presumably due to finding available space as the process memory map becomes fragmented.

To eliminate the memory-residency overheads, we use Flash's mapped-file cache bookkeeping as the sole heuristic for guessing memory residency. We eliminate all `mmap/mincore/munmap` calls but keep track of what pieces of files have been recently accessed. Sizing the cache conservatively with respect to main memory, we save CPU overhead but introduce a small risk of having the main process block. The CPU savings of this approach is substantial, allowing us to reach 620 simultaneous connections and a 2GB data set size.

5.7 Dynamic Content Interface

We take advantage of DeBox's ability to separate the kernel time consumption by call site to determine that although the `read()` system call is used by the main process, the helpers, and all of the CGI processes, the single call site responsible for most of the time is where the main process reads from the CGIs. Flash uses a persistent CGI interface similar to FastCGI [32] to reuse CGI processes when possible, and this mechanism communicates over pipes.

Our measurements show that this call site consumes 20% of all kernel time, (176 seconds out of 891 seconds total). Writing the request to the CGI processes is much smaller, requiring only 24.3 seconds of system call time. This level of detail demonstrates the power of making performance a first-class result, since existing kernel profilers would not have been able to separate the time for the `read()` call by call site. By modifying our CGI interface slightly, we allow the main process to write the HTTP response to the client, and then pass the socket to the CGI to let it write directly. This change allows us to reach 710 simultaneous connections, and a 2.35GB data set size.

time	label	kernel file	line
6492	sfbufa	kern/uipc_syscalls.c	1459
702	getblk	kern/kern_lock.c	182
984544	biord	kern/vfs_bio.c	2724

Table 6: **New blocking measurements of** `sendfile()`

5.8 Optimizing sendfile()

We return our focus to the `sendfile()` system call for a variety of reasons: we had noted worse performance than `writev()`, we had seen some blocking at the label "sfpbsy" in Table 4, and our replacement of `mincore()` with a heuristic may cause more blocking. New PerSleepInfo measurements of the blocking behavior of `sendfile()` are shown in Table 6.

The resource label "sfbufa" indicates that the kernel has exhausted the sendfile buffers used to map filesystem pages into kernel virtual memory. We confirm that manually increasing the number of buffers eliminates this problem in our test. However, based on the results of previous copy-avoidance systems [18, 35], we opt instead to implement recycling of kernel virtual address buffers. With this change, many requests to the same file do not cause multiple mappings, and eliminates the associated virtual memory and physical map (pmap) operations. Caching these mappings may temporarily use more wired memory than no caching, but the reduction in overhead and address space consumption outweighs the drawbacks.

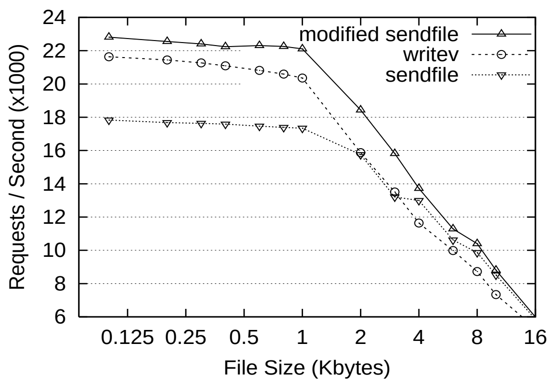
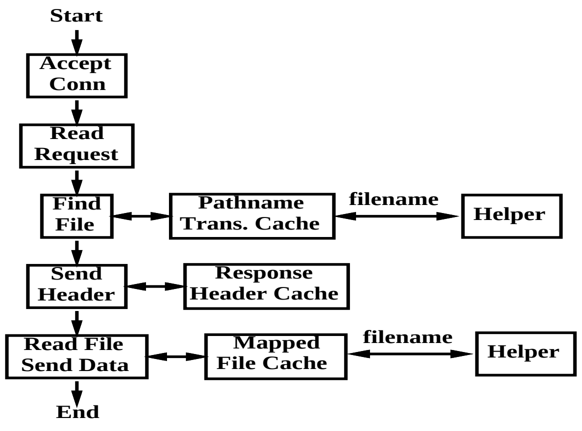


Figure 7: **Microbenchmark performance comparison of writev, sendfile, and modified sendfile:** In this test, all clients request a single file at full speed using persistent connections.

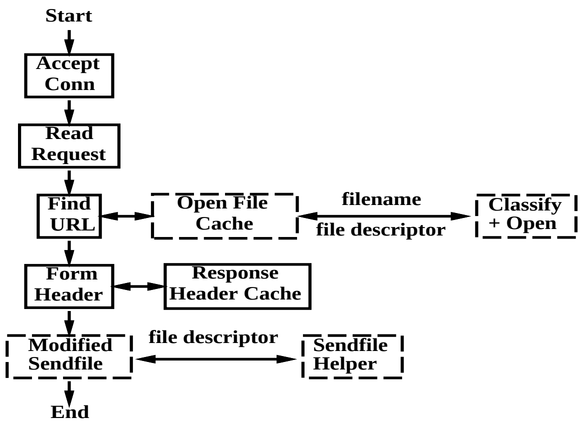
The other two resource labels, "getblk" and "biord", are related to disk access initiated within `sendfile()` when the requested pages are not in memory. Even though the socket being used is nonblocking, that behavior is limited only to network buffer usage. We introduce a variant `sendfile()` call with slightly different semantics, which returns a different `errno` value if disk blocking would occur. This change allows us to achieve the same effect as we had with `mincore()`, but with much less CPU overhead. We may optionally have the read helper process send data directly back to the client on a filesystem cache miss, but have not implemented this optimization.

However, even when `sendfile()` does not block, we observe no performance gain over `writev()`, and we find that the problem stems from handling small writes. HTTP static content responses consist of a small header followed by file data. Using `writev()` allows aggregation of the header and the first portion of the body data into one packet, benefiting small file transfers. In SpecWeb99, 35% of all static requests are for files 1KB or smaller.

The FreeBSD `sendfile()` call includes parameters specifying headers and trailers to be sent with the data, whereas the Linux implementation does not. Linux introduces a new socket option to "cork" the TCP stream so that HTTP header data sent via `write()` can be combined with zero-copy packet data. While FreeBSD's "monolithic" approach provides enough information to avoid sending a separate header, its implementation sends the header using a kernel version of `writev()`, thus generating a separate packet for the response header. We improve this implementation by creating an mbuf chain using the header and body data before sending it to lower levels of the network stack. This change generates fewer packets,



(a) Original Architecture



(b) New Architecture

Figure 8: **Architectural changes:** The architecture is greatly simplified by using file descriptor passing and eliminating mapped file caching. Modified components are indicated with a dashed box.

improving performance and network latency. Results of these changes on a microbenchmark are shown in Figure 7. With the `sendfile()` changes, we are able to achieve a SpecWeb99 score of 820, with a data set size of 2.7GB.

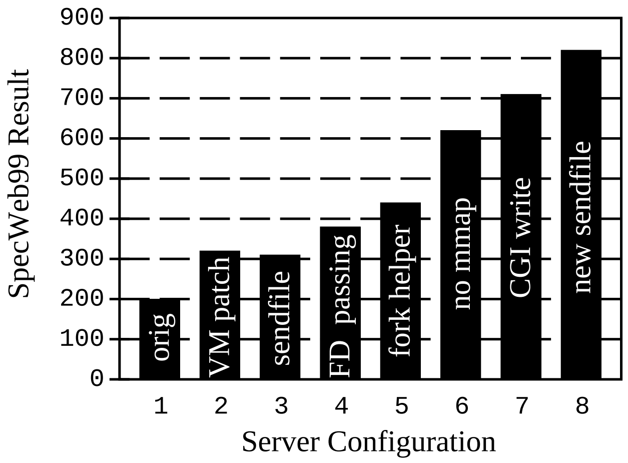


Figure 9: **SpecWeb99 summary:** 1. Original 2. VM patch 3. Using sendfile() 4. FD-passing helpers 5. Fork helper 6. Eliminate mmap 7. New CGI interface 8. New sendfile()

6 Case Study Summary

By addressing the interaction areas identified by DeBox, we achieve a factor of four improvement in our SpecWeb99 score, supporting four times as many simultaneous connections while also handling a data set that almost three times as large as the physical memory of our machine. The SpecWeb99 results of our modifications can be seem in Figure 9, where we show the scores for all of the intermediate modifications described in this paper. Our final result of 820 compares favorably to published SpecWeb99 scores, though no directly-comparable systems have been benchmarked. We outperform all uniprocessor systems with similar memory configurations – the highest score for a system with less than 2GB of memory is 575.

Most of our changes are portable architectural modifications to the Flash Web Server, including (1) passing file descriptors between the helpers and the main process to avoid most disk operations in the main process, (2) introducing a new `fork()` helper to handle forking CGI requests, (3) eliminating the mapped file cache, and (4) allowing CGI processes to write directly to the clients instead of writing to the main process. Figure 8 shows the original and new architectures of the static content path for the server.

The changes we make to the operating system focus on `sendfile()`, including (1) changing the semantics to indicate when blocking on disk would occur, (2) caching kernel address space mapping to avoid unnecessary physical map operations, and (3) sending headers and file data in a single mbuf chain to avoid multiple packets for small responses. Additionally, we apply a virtual memory system patch that ultimately is superfluous since we remove the memory-mapped file cache.

7 Latency

Since we identify and correct many sources of blocking, we are interested in the effects of our changes on server latency. We first compare the effect of our changes on a SpecWeb99 workload, and then reproduce workloads used by other researchers in studying static content latencies. In all cases, we compare latencies using a workload below the maximum of the slowest server configuration under test.

On the SpecWeb99 workload, we find that mean response time is reduced by a factor of four by our changes. The cumulative distribution of latencies can be seen in Figure 10. We use 300 simultaneous connections, and compare the new server with the original Flash running on a

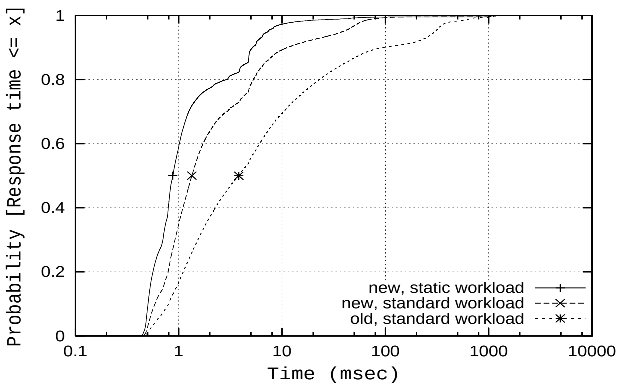


Figure 10: **Latency summary for 300 SpecWeb99 connections**

patched VM system. Since 30% of the requests are for longer-running dynamic content, we also test the latencies of a SpecWeb99 test with only static requests. The mean of this workload is 7.08 msec, lower than the 10.6 msec mean for the new server running the complete workload. This difference suggests that further optimization of dynamic content interface may lead to even better performance. To compare the difference between static and dynamic request handling, we calculate the 5th, 50th, and 95th percentiles of the latencies for requests on the SpecWeb99 workload. These results are shown in Table 7, and indicate that dynamic content is served at roughly half the speed of its static counterpart. The latency difference between the new server and the original Flash on this test is not as large as expected because the working set still fits in physical memory.

	5%(ms)	50%(ms)	95%(ms)	mean(ms)
static	0.51	1.45	59.81	9.92
dynamic	0.99	2.83	91.31	12.19

Table 7: **Separating SpecWeb99 static and dynamic latencies**

To determine our latency benefit on a more disk-bound workload and to compare our results with those of other researchers, we construct a static workload similar to the one used to evaluate the Haboob server [46]. In this workload, 1020 simulated clients generate static requests to a 3.3GB data set. To avoid overloading the slowest server, the request rate is fixed at 170 requests per second. Persistent connections are used, with clients issuing 5 requests over a single connection before closing it. Our test environment differs from that used to evaluate Haboob in the following ways: our system has only 1GB of memory versus 2GB, we have a single 933 MHz processor versus four 500 MHz processors, and we are using FreeBSD versus Linux.

We compare several configurations to determine the latency benefits and the impact of parallelism in the server. We run the new and original versions of Flash with a single instance and four instances, to compare uniprocessor configurations with what would be expected on a 4-way SMP.

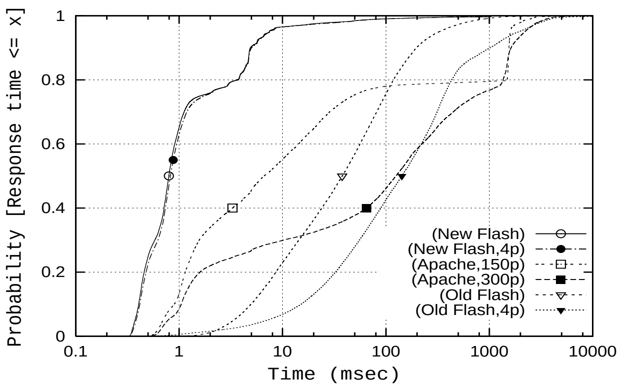


Figure 11: **Response latencies for the 3.3GB static workload**

	5% (ms)	median (ms)	95% (ms)	mean (ms)
New Flash	0.37	0.79	7.45	7.56
New Flash, 4p	0.38	0.82	7.51	7.72
Old Flash	3.36	37.59	326.40	92.37
Old Flash, 4p	7.05	142.65	1924.42	420.85
Apache 150p	0.70	6.64	1599.50	360.62
Apache 300p	0.78	124.98	2201.63	545.93

Table 8: **Summaries of the static workload latencies**

We also run Apache with 150 and 300 server processes.

The results are given in Figure 11 and Table 8 and show the response time of our new server under this workload exhibits improvements of more than a factor of twelve in mean response time, and a factor of 47 in median latency. With four instances, the difference are a factor of 54 in mean response time and 174 in median time. We measure the maximum capacities of the servers when run in infinite-demand mode, and these results are shown in Table 9. While the throughput gain from our optimizations is significant, the scale of gain is much lower than the SpecWeb99 test, indicating that our latency benefits do not stem purely from extra capacity.

We also observe that all servers tested show latency degradation when run with more processes, though the effect is much lower for our new server. This observation is in line with the self-interference we described earlier between the helpers and the main Flash process, but is observed even with Apache. We confirm this by using DeBox to measure the number of sleeps when running Apache. With 150 processes, Apache blocks 3667 times per second, and this increases to 3994 times per second at 300 processes.

This result suggests that excess parallelism, where server designers use parallelism for convenience, may actually de-

New Flash		Old Flash		Apache	
1p	4p	1p	4p	150p	300p
326.4	308.6	264.5	221.1	210.6	201.5

Table 9: **Server static workload capacities (Mb/s)**

grade performance noticeably. This observation may explain the latency behavior reported for Haboob [46]. The median latency shown for Flash in that paper is approximately 50ms, comparable to the 37.59ms median we measure. The mean latency given for Haboob is 547ms and its median is approximately 500-600ms. In comparison, our mean latency for the new version of Flash is 7.56ms and our median is 0.82ms, suggesting our latencies are 70-500 times lower than Haboob.

8 Related Work

In this section, we discuss other related work not already covered in our discussion in Section 2. The idea of observing kernel behavior to improve performance has appeared in many different forms. We share similarities with Scheduler Activations [5] in observing scheduler activity to optimize application performance, and with Marsh et al. [27], who make user-level threads first-class with kernel support. Our goals differ, since we are more concerned with understanding why blocking occurs rather than reacting to it during a system call. Our modification of `sendfile()` to indicate blocking is patterned on non-blocking sockets, but it could be used in other system calls as well. In a similar vein, RedHat has applied for a patent on a new flag to the `open()` call, which causes it to fail if the necessary metadata is not in memory [30].

Our observations on excess parallelism and its impact on latency may impact server design. Performance studies of the Harvest Cache [12] established the suitability of event-driven designs for network servers, and the Flash server demonstrated how to avoid some disk-related blocking [34]. Schmidt and Hu [40] performed much of the early work in studying threaded architectures for improving server performance. A similar architecture was used by Welsh et al. [46] to support concurrency and provide scheduling behavior. Larus and Parkes [25] demonstrate that such scheduling can also be performed in event-driven architectures. Qie et al. [38] show that such architectures can also be protected against denial-of-service attacks. Adya et al. [1] discuss the unification of the two models. We believe that DeBox can be used to identify problem areas in other servers and architectures, as our latency measurements of Apache suggest.

Most of the changes in our case study modified the server code rather than the operating system. This observation may indicate that the incorporation of previous OS research into mainstream operating systems has been successful, or that problem avoidance is equally viable as kernel modification. Extensible kernels [10, 19, 20, 41] may provide opportunities for applications to “fix” problems that can not easily be avoided, such as our implementation changes of `sendfile()`. Likewise, conveniences, such as our semantic change of its behavior, would also become more attractive, but these may only be appropriate for trusted applications.

9 Conclusion

This paper presents the design, implementation and evaluation of DeBox, an effective approach to provide more OS transparency, by exposing system call performance as a first-class result via in-band channels. DeBox provides direct performance feedback from the kernel on a per-call basis, enabling programmers to diagnose kernel/user interactions correlated with user-level events. Furthermore, we believe that the ability to monitor behavior on-line provides programmatic flexbility of interpreting and analyzing data not present in other approaches.

Our case study using the Flash Web Server with the SpecWeb99 benchmark running on FreeBSD demonstrates the power of DeBox. Addressing the problematic interactions and optimization opportunities discovered using DeBox improves our experimental results an overall factor of four in SpecWeb99 score, despite having a data set size nearly three times as large as our physical memory. Furthermore, our latency analysis demonstrates gains between a factor of 4 to 47 under various conditions. Further results show that fixing the bottlenecks identified using DeBox also mitigates most of the negative impact from excess parallelism in application design.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative tasking without manual stack management. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[2] W. Akkerman. strace homepage. http://www.wi.leidenuniv.nl/wichert/strace/.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, June 1997.

[4] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 1–14, Saint-Malo, France, Oct. 1997.

[5] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[6] Apache Software Foundation. The Apache Web server. http://www.apache.org/.

[7] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. of the 19th ACM Symp. on Operating System Principles*, pages 43–56, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[8] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[9] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX 1999 Annual Technical Conference*, pages 253–265, Monterey, CA, June 1999.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Operating System Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[11] C. Blake and S. Bauer. Simple and general statistical profiling with pct. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[12] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125, 1995.

[13] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *ACM SIGMETRICS Conference*, pages 214–224, Seattle, WA, June 1997.

[14] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[15] N. Burnett, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[16] E. Cota-Robles and J. P. Held. A comparison of windows driver model latency performance on windows NT and windows 98. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation*, pages 159–172, New Orleans, LA, Feb. 1999.

[17] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, Oct. 1996.

[18] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 189–202, Asheville, NC, Dec. 1993.

[19] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond microkernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan., June 1992.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symp. on Operating System Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[21] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, Massachusetts, June 1982.

[22] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. In *Software- Practice and Experience*, pages 671–685, 1983.

[23] Intel. Vtune Performance Analyzers Homepage. http://developer.intel.com/software/products/vtune/index.htm.

[24] M. B. Jones and J. Regehr. The problems you're having may not be the problems you think you're having: Results from a latency study of windows nt. In *7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[25] J. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *USENIX 2002 Annual Technical Conference*, pages 103–114, Monterey, CA, June 2002.

[26] J. Lemon. Kqueue: A generic and scalable event notification facility. In *FREENIX Track: USENIX 2001 Annual Technical Conference*, pages 141–154, Boston, MA, June 2001.

[27] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proc. of the 13th ACM Symp. on Operating System Principles*, pages 110–121, Pacific Grove, CA, Oct. 1991.

[28] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*, pages 279–294, San Diego, CA, June 1996.

[29] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[30] I. Molnar. Method and apparatus for atomic file look-up. United States Patent Application #20020059330, May 16, 2002.

[31] E. Nahum. Deconstructing SPECweb99. In *7th International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, CO, Aug. 2002.

[32] Open Market. FastCGI. http://www.fastcgi.com/.

[33] OProfile. A system profiler for linux. http://oprofile.sourceforge.net/.

[34] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[35] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[36] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, 1993.

[37] L. K. Puthiyedath, E. Cota-Robles, J. Keys, and J. P. H. Anil Aggarwal. The design and implementation of the intel real-time performance analyzer. In *Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, San Jose, CA, Sept. 2002.

[38] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[39] T. Romer, G. V. D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *USENIX Windows NT Workshop*, pages 1–8, 1997.

[40] D. C. Schmidt and J. C. Hu. Developing flexible and high-performance Web servers with frameworks and patterns. *ACM Computing Surveys*, 32(1):39, 2000.

[41] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996.

[42] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[43] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.spec.org/cpu2000.

[44] Standard Performance Evaluation Corporation. SPEC Web 96 & 99 Benchmarks. http://www.spec.org/osg/ web96/ and http://www.spec.org/osg/web99/.

[45] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation*, pages 117–130, New Orleans, LA, Feb. 1999.

[46] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of the 19th ACM Symp. on Operating System Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[47] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[48] C. X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 15–26, Saint-Malo France, Oct. 1997.