USENIX Association

Proceedings of the General Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

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 operating system intensive applications, the ability of designers to understand system call performance behavior is essential to achieving high performance. Conventional performance tools, 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 applications via *in-band* channels on a *per-call* basis presents opportunities for performance analysis and 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 their actions 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 and 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 OS. Addressing these issues as well as other optimization opportunities generates an overall factor of four improvement in our SpecWeb99 score, throughput gains on other benchmarks, and latency reductions ranging from a factor of 4 to 47.

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 developing new system calls that are more closely tailored to the needs of particular applications. In particular, zero-copy I/O [17, 31] and scalable event delivery [9, 10, 23] 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 [32], have also been proposed and implemented. Some system calls, such as `madvise()`, provide usage 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 [30] 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] manages 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, latency, or both. 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 call. The term *first-class result* specifies that it gets treated the same as other results, such as errno and the system call return value, instead of having to be explicitly requested via other system or library calls. The term *in-band* specifies that it is returned to the caller immediately, instead of being logged or processed by some other monitoring processes. While it is much larger and more detailed than the errno global variable, they are conceptually similar. Simply monitoring at the system call boundary, the scheduler, page fault handlers, and function entry and exit is sufficient to provide detailed information about the inner working of the operating system. This approach not only eliminates guesswork about what happens during call processing, but also 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.

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 under contention, 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 [39]. Using DeBox, we are able to diagnose a series of problematic interactions between the server and the operating system on this benchmark. The resulting system shows an overall factor of four improvement in SpecWeb99 score, throughput gains on other benchmarks, and latency reductions ranging from a factor of 4 to 47. Most of the issues are addressed by application redesign and the resulting system is portable, as we demonstrate by showing improvements on Linux. Our kernel modifications, optimizing of the sendfile() system call, have been integrated into FreeBSD.

DeBox is specifically designed for performance analysis of the interactions between the OS and applications, especially in server-style environments with complex workloads. Its combination of features and flexibility is novel, and differentiates it from other profiling-related approaches. However, it is not designed to be a general-purpose profiler, since it currently does not address applications that spend most of their time in user space or in the “bottom half” (interrupt-driven) portion of the kernel.

The rest of this paper is organized as follows. In Section 2 we motivate the approach of DeBox. The detailed design and implementation are described in Section 3. We describe experimental setup and workloads in Section 4, then show a case study using DeBox to analyze and optimize the Flash Web Server in Section 5. Section 6 contains further experiments on latency and Section 7 demonstrates the portability of our optimizations. We discuss related work in Section 8 and conclude in Section 9.

2 Design Philosophy

DeBox is designed to bridge the divide in performance analysis across the kernel and user boundary by exposing kernel performance behavior to user processes, with a focus on server-style applications with 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.

We present our observations about performance analysis for server applications as below. While some of these measurements could be made in other ways, we believe that DeBox’s approach is particularly well-suited for these environments. Note that replacing any of the existing tools is an explicit non-goal of DeBox, nor do we believe that such a goal is even feasible.

High overheads hide bottlenecks. The cost of the debugging tools may artificially stress parts of the system, thus masking the real bottleneck at higher load levels. Problems that appear only at high request rates may not appear when a profiler causes an overall slowdown. Our tests show that for server workloads, kernel gprof has 40% performance degradation even when low resolution profiling is configured. Others tracing and event logging tools generate large quantities of data, up to 0.5MB/s in Linux Trace Toolkit [42]. For more demanding workloads, the CPU or filesystem effects of these tools may be problematic.

We design DeBox not only to exploit hardware performance counters to reduce overhead, but also to allow users to specify the level of detail to control the overall costs. Furthermore, by splitting the profiling policy and mechanism in DeBox, applications can decide how much effort to expend on collecting and storing information. Thus, they may selectively process the data, discard redundant or trivial information, and store only useful results to reduce the costs. Not only does this approach make the cost of profiling controllable, but one process desiring profiling does not affect the behavior of others on the system. It affects only its own share of system resources.

User-level timing can be misleading. 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 for in-memory data. As a result, the high peaks on this graph are troubling, since they suggest the server is blocking. A similar measurement using getrusage() also falsely implies the same. Even though the measurement calls immediately precede and follow the system call, heavy system activity causes the scheduler to preempt the process in that small window.

In DeBox, we integrate measurement into the system call process, so it does not suffer from scheduler-induced measurement errors. The DeBox-derived measurements of the same call are shown in Figure 2, and do not indicate such sharp peaks and blocking. Summary data for sendfile and accept (in non-blocking mode) are shown in Table 1.

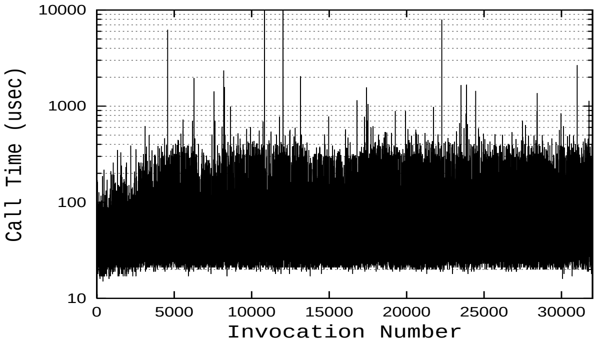

Figure 1: User-space timing of the `sendfile` call on a server running the SpecWeb99 benchmark – note the sharp peaks, which may indicate anomalous behavior in the kernel.

	`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.

Statistical methods miss infrequent events. Profilers and monitoring tools may only sample events, with the belief that any event of interest is likely to take "enough" time to eventually be sampled. However, the correlation between frequency and importance may not always hold. Our experiments with the Flash web server indicate that adding a 1 ms delay to one out of every 1000 requests can degrade latency by a factor of 8 while showing little impact on throughput. This is precisely the kind of behavior that statistical profilers are likely to miss.

We eliminate this gap by allowing applications to examine every system call. Applications can implement their own sampling policy, controlling overhead while still capturing the details of interest to them.

Data aggregation hides anomalies. Whole-system profiling and logging tools may aggregate data to keep completeness and reduce overhead at the same time. This approach makes it hard to determine which call invocation experienced problems, or sometimes even which process or call site was responsible for high-overhead calls. This problem gets worse in network server environments where the systems are complex and large quantities of data are generated. It is not uncommon for these applications to have dozens of system call sites and thousands of invocations per second. For example, the Flash server consists of about 40 system calls and 150 calling sites. In these conditions, either discarding call history or logging full events is infeasible.

By making performance information a result of system calls, developers have control over how the kernel profiling is performed. Information can be recorded by process and by call site, instead of being aggregated by call number inside the kernel. Users may choose to save accumulated results, record per-call performance history over time, or fully store some of the anomalous call trace.

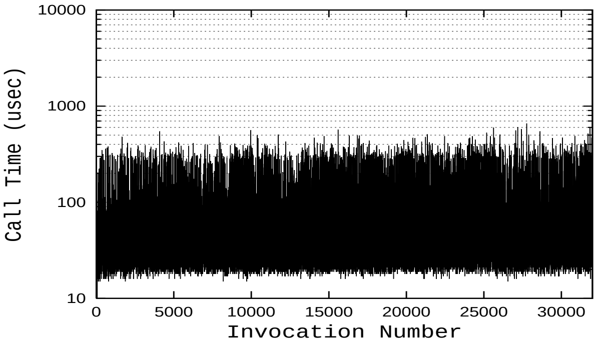

Figure 2: The same system call measured using DeBox shows much less variation in behavior.

Out-of-band reporting misses useful opportunities. As the kernel-user boundary becomes a significant issue for demanding applications, understanding the interaction between operating systems and user processes becomes essential. Most existing tools provide measurements out-of-band, making online data processing harder and possibly missing useful opportunities. For example, the online method allows an application to `abort()` or record the status when a performance anomaly occurs, but it is impossible with out-of-band reporting.

When applications receive performance information tied to each system call via in-band channels, they can choose the filtering and aggregation appropriate for the program's context. They can easily correlate information about system calls with the underlying actions that invoke them.

3 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 compare overhead for DeBox support and active use versus an unmodified kernel. Examples of how to fully use DeBox and what kinds of information it provides are deferred to the case study in Section 5.

3.1 User-Visible Portion

The programmer-visible interface of DeBox is intentionally simple, since it consists of some monitoring data structures and a new system call to enable and disable data gathering. Figure 3 shows DeBoxInfo, the data structure that handles the DeBox information. 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 specify how much per-call performance information it desires.

```
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

At first glance, the DeBoxInfo structure appears 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 process specifies limits on how much of it is used. Most of the space is consumed by two arrays, PerSleepInfo and CallTrace. 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.

3.2 In-Kernel Implementation

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

Since DeBox returns performance information when each system call finishes, the system call entry and 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 records information while the system call executes, avoiding many small copies between kernel and user. Prior to system call return, the requested 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 it. However, since the DeBoxInfo structure can contain other arrays, more detailed page fault information can be added 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. Kernel routines invoking `tsleep()` 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 and 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()`

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

PerSleepInfo[0]:
           1270  /* # occurrences */
         723903  /* time blocked, microsecs */
           biowr  /* resource label */
 kern/vfs_bio.c  /* file where blocked */
           2727  /* line where blocked */
              1  /* # processes on entry */
              0  /* # processes on exit */

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

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

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 a system call 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 the current time and function name when it started and finished, similar to what call graph profilers use. The gcc compiler allows entry and 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 and exit macros into reachable functions. The entry macro pushes current function address and time in a temporary stack. The exit macro pops out the function address, calculates the wall clock time, and records these information in 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 show what kind of information is provided in DeBox, we give a sample output in Figure 4. We memory-map a 10MB file, and use 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 used about 3.6 seconds of wall-clock time. It incurred 989 page faults, and blocked in two unique places in the kernel. The first PerSleepInfo element 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", read of a special file. The writes blocked for roughly 0.7 seconds, and the reads for 2.7 seconds.

3.3 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.

call name or read size	base time	DeBox without call trace		DeBox call trace	
		off	on	off	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 uses an unmodified system. All times are in microseconds

We separate the measurement for call history tracing since we do not expect it will be activated continuously. These numbers show 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 to the whole system is even lower. The performance impact with DeBox disabled, indicated by the 3rd column, is virtually unnoticeable. The cost of supporting call tracing, shown in the 5th 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 integrate it with the function prologue and epilogue code. We expect that we can reduce this overhead, along with the overhead of using the call tracing, with optimization.

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

Table 3: DeBox macrobenchmark overheads

Since microbenchmarks do not indicate what kinds of slowdowns may be typically observed, we provide some macrobenchmark results to give some insight into these costs in Table 3. The three systems tested are: an unmodified system, one with only "basic" DeBox without call trace support, 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% to roughly 3% in the kernel build. We expect that many environments will tolerate this overhead in exchange for the flexibility provided by DeBox.

4 Experimental Setup & Workload

We describe our experimental setup and the relevant software components of the system in this section. All of our experiments, except for the portability measurements[1], are performed on a uniprocessor server running FreeBSD 4.6, with a 933MHz Pentium III, 1GB of memory, one 5400 RPM Maxtor IDE disk, 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 SpecWeb99.

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 number of helper processes 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 [30]. We run with logging disabled to simplify comparison with Apache, where enabling logging degrades performance noticeably.

We focus on the SpecWeb99 benchmark, an industry-standard test of the overall scalability of Web servers under realistic conditions. It is designed by SPEC, the developers of the widely-used SpecCPU workloads [38], and is based on traffic at production Web sites. Although not common in academia, it is the *de facto* standard in industry [27], with over 190 published results, and is different from most other Web server benchmarks in its complexity and requirements. It measures scalability by reporting the number of simultaneous connections the server can handle while meeting a specified quality of service. The data set and working set sizes 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 Case Study

In this section, we demonstrate how DeBox's features can be used to analyze and optimize the behavior of the Flash Web Server. 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 conventional solutions, such as throwing more resources at the problem, may exacerbate the problem. Our optimizations quadruple our SpecWeb99 score and also sharply decrease latency.

5.1 Initial experiments

Our first run of SpecWeb99 on the publicly 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 on comparable hardware using TUX, an in-kernel Linux-only HTTP server. At 200 simultaneous connections, the dataset size is roughly 770MB, 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 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 lists with splay trees, 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 1.13GB, slightly exceeding our physical memory.

Once the `mincore()` problem is addressed, we still appear to be CPU-bound, and suspect the data copying is the bottleneck. So we update the Flash server to use the zero-copy I/O system call, `sendfile()`. 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()`, which is a scalable event delivery mechanism recently incorporated into FreeBSD. After these changes, we are not surprised by the drop in CPU utilization, but are surprised that the SpecWeb99 score drops to 300.

5.2 Successive refinement of detail

With the server exhibiting idle CPU time but an inability to meet SpecWeb99's quality-of-service requirements, an obvious candidate is blocking system calls. However, Flash's main process is designed to avoid blocking. We tried tracing the problem using existing tools, but found they suffered from the problems discussed in section 2. These experiences motivated the creation of DeBox.

[1] The Linux kernel crashes on our existing server hardware

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

The DeBox structures provide various levels of detail, allowing applications to specify what to measure. A typical use would first collect the basic DeBoxInfo to observe anomalies, then enable more details to identify the affected system calls, invocations, and finally the whole call trace.

We first use DeBox to get the blocking information, which is stored in the PerSleepInfo field. The PerSleepInfo data shows seven different system calls blocking in the kernel, and examination of the resource labels (wmesg) shows four reasons for blocking. These results are shown in Table 4, where each column header shows the resource label 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, which is not surprising since our data set exceeds the physical memory of the machine.

The finest-grained kernel information is provided in the CallTrace structure, and we enable this level of detail once the PerSleepInfo identifies possible candidates. The main process should only be accessing cached data, so the fact that it blocks on disk-related calls is puzzling. For portability, the main process in Flash uses the helpers to demand-fetch disk data and metadata into the OS caches, and repeats the operation immediately after the helpers have completed loading, assuming that the recently loaded information will prevent it from blocking. Observing the full CallTrace of some of these blocking invocations shows the blocking is not caused by disk access, but contention on filesystem locks. Combining the blocking information from helper processes reveals that when the main process blocks, the helpers are operating on similarly-named files as the main process. We solve this problem by having the helpers return open file descriptors using `sendmsg()`, eliminating duplication of work in the main process. With this change, we are able to handle 370 simultaneous connections from SpecWeb99, with a dataset size of 1.28GB.

5.3 Capturing rare anomaly paths

We find that the `sendmsg()` change solves most of the filesystem-related blocking. However, one `open()` call in Flash still shows occasional blocking at the label "biord" (reading a disk block), but only after the server has been running for some time and under heavy workloads. Only revealing which call induced the problem may not suffice a complete picture, because the reason of invoking that call is unclear. In a system with multiple identical system calls, existing tools do not have an efficient way to find which one causes the problem and the calling path involved.

Because DeBox information is returned in-band, the user-space context is also available once kernel performance anomaly is detected. On finding a blocking invocation of `open()`, we capture the path through the user code by calling `abort()` and using `gdb` to dump the stack[2]. This approach 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-mapped 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 and has the metadata cached. When Flash calls `open()` to access the file in this circumstance, 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. After fixing this bug, we are able to handle 390 simultaneous connections, and a 1.34GB dataset.

5.4 Tracking call histories

With all blocking eliminated and with a much higher request rate, we return to the issue of CPU consumption. By storing the CallTime field of each system call, we can track per-call performance by invocation, both to observe trends and to identify time-related problems. Traditional profiling tools usually report average CPU consumption of each function, thus hiding any performance trends. User space timing functions may catch the general trend in spite of the measurement error, but involve much more work to track each system call and find the problematic ones.

5.4.1 Process creation overhead

By recording all CPU time values, we find that the largest call times are for the `fork()` system call and that its cost grows with the number of invocations, approaching 130 msec. Figure 5 shows the per-call time as a function of invocation. We observe that `fork()` time increases as the program runs, starting as low as 0.3 msec. These calls stem from the SpecWeb99 workload's requirement that 0.15% of the requests be handled by forking new processes.

A full call trace indicates that `fork()` spends the bulk of its time copying file descriptors and VM map entries (for mapped regions). Rather than changing the implementation of `fork()`, we opt to slightly modify the Flash archi-

[2] Alternately, we could invoke `fork()` followed by `abort()` to keep the process running while still obtaining a snapshot, or we could record the call path manually

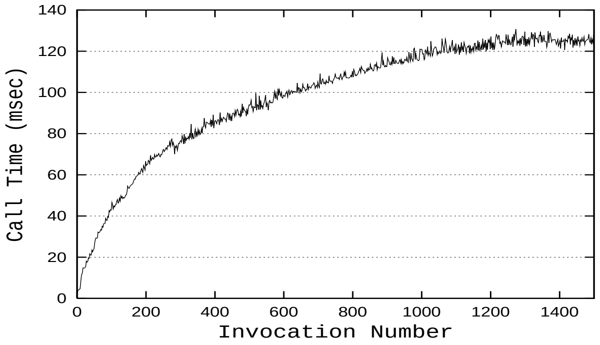

Figure 5: Call time of `fork()` as a function of invocation

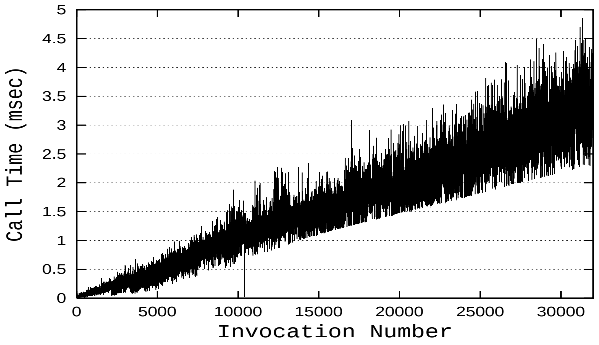

Figure 6: Call time of `mmap()` as a function of invocation

tecture. We introduce a new helper process that is responsible for creating 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 1.50GB dataset size.

5.4.2 Memory lookup overhead

Though the dataset size now exceeds physical memory by over 50%, the system bottleneck remains CPU. Examining the time consumption of each system call again reveals that most time is being spent in memory residency checking. Though our modified Flash uses `sendfile()`, it uses `mincore()` to determine memory residency, which 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 avoid 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, and 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 change is substantial, allowing us to reach 620 connections (2GB dataset).

5.5 Profiling by call site

We take advantage of DeBox's flexibility by separating the kernel time consumption based on call site rather than call name. We are interested in the cost of handling dynamic content since SpecWeb99 includes 30% dynamic requests which could be processed by various interfaces. Flash uses a persistent CGI interface similar to FastCGI [28] to reuse CGI processes when possible, and this mechanism communicates over pipes. Although the `read()` and `write()` system calls are used by the main process, the helpers, and all of the CGI processes, we measure the overhead of only those involved in communication with CGI processes.

Our measurements show that the single call site responsible for most of the time is where the main process reads from the CGIs, consuming 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()` calls by call sites. By modifying our CGI interface slightly, the main process writes only the HTTP header to the client, and passes the socket to the CGI application to let it write the data directly. This change allows us to reach 710 connections (2.35GB dataset).

5.6 Other optimization opportunities

By replacing our exact memory residency check with a cheaper heuristic, we gain performance, but introduce blocking into the `sendfile()` system call. New PerSleepInfo measurements of the blocking behavior of `sendfile()` are shown in Table 5.

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 5: New blocking measurements of `sendfile()`

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 increasing the number of buffers during boot time may mitigate this problem in our test. However, based on the results of previous copy-avoidance systems [17, 31], 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.

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 new flag to `sendfile()` so that it 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 with blocking eliminated, we find performance barely changes when using `sendfile()` versus `writev()`, and we find that the problem stems from handling small writes. HTTP responses consist of a small header followed by file data. The `writev()` code aggregates 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, TCP_CORK, to delay transmission until full packets can be assembled. While FreeBSD's "monolithic" approach provides enough information to avoid sending a separate header, its implementation uses a kernel version of `writev()` for the header, thus generating an extra packet. 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, 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 dataset size of 2.7GB.

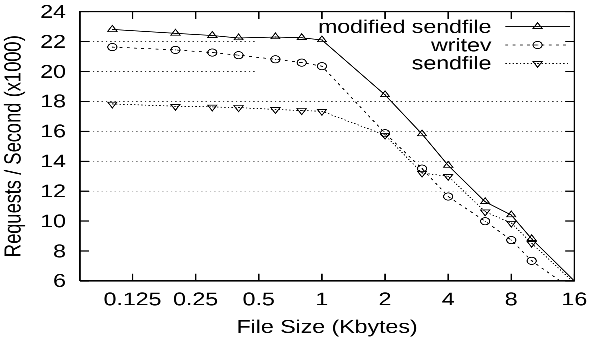

Figure 7: Microbenchmark performance comparison of writev, sendfile, and modified sendfile – In this test, all clients request a single file at full speed using persistent connections.

5.7 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 seen in Figure 8, where we show the scores for all of the intermediate modifications we made. 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 but using other server software – the highest score for a system with less than 2GB of memory is 575.

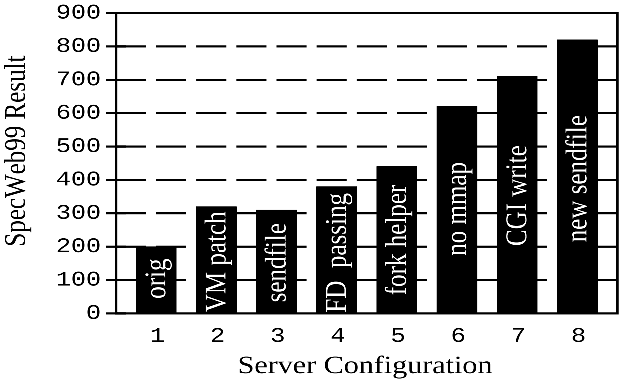

Figure 8: SpecWeb99 summary – 1. Original 2. VM patch 3. Using sendfile() 4. FD-passing helpers 5. Fork helper 6. Eliminate mmap 7. New CGI interface 8. New sendfile()

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 9 shows the original and new architectures of the static content path for the server.

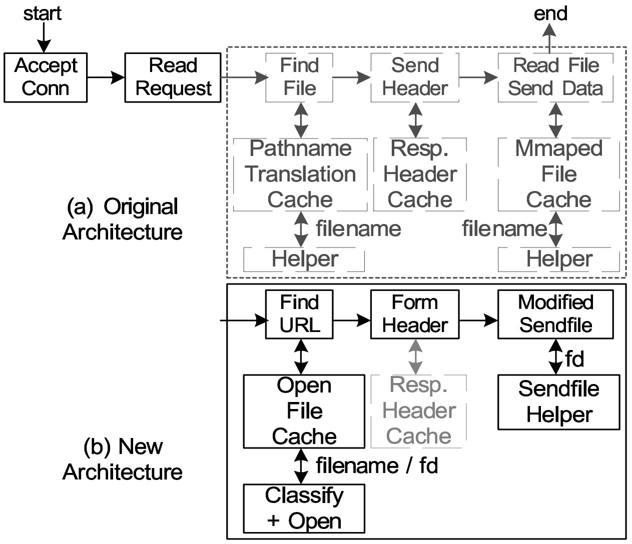

Figure 9: Architectural changes – The architecture is greatly simplified by using file descriptor passing and eliminating mapped file caching. Modified components are indicated with dark boxes.

The changes we make to the operating system focus on `sendfile()`, including (1) adding a new flag and return value 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. We have provided our modifications to the FreeBSD developer group and all three optimizations have been incorporated into FreeBSD.

6 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 the 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.

6.1 SpecWeb99 workload

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 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 7.1 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 handling may lead to even better performance. To compare the difference between static and dynamic request handling, we calculate the 5^{th}, 50^{th}, and 95^{th} percentiles of the latencies for requests on the SpecWeb99 workload. These results are shown in Table 6, 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.

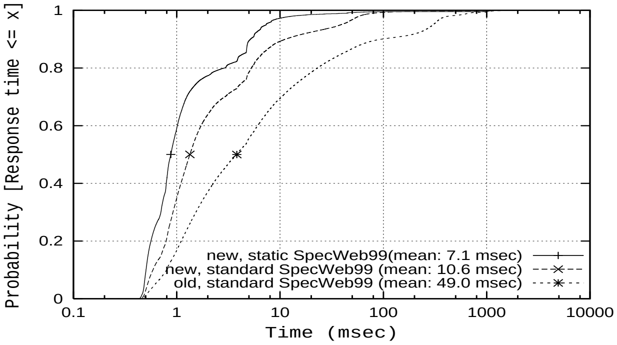

Figure 10: Latency summary for 300 SpecWeb99 connections

	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 6: Separating SpecWeb99 static and dynamic latencies

6.2 Diskbound static workload

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 [41]. In this workload, 1020 simulated clients generate static requests to a 3.3GB data set. Persistent connections are used, with clients issuing 5 requests per connection before closing it. To avoid overload, the request rate is fixed at 2300 requests/second, which is roughly 90% of the slowest server's capacity.

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. We also run Apache with 150 and 300 server processes.

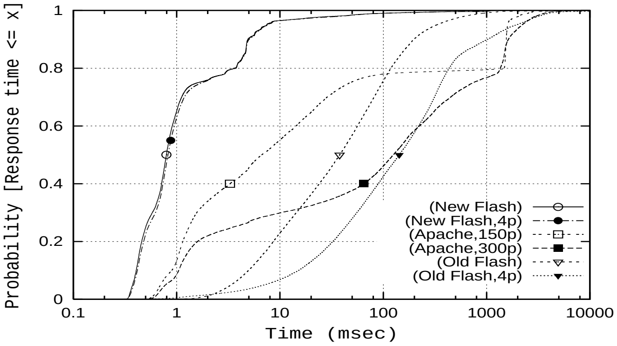

Figure 11: Response latencies for the 3.3GB static workload

	5%	median	95%	mean
	(ms)	(ms)	(ms)	(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 7: Summaries of the static workload latencies

The results, given in Figure 11 and Table 7, 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 differences 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 8. While the throughput gain from our optimizations is significant, the scale of gain is much lower than the SpecWeb99 test, in-

data set	Apache	Old Flash	New Flash
500MB	240.3	485.2	660.9
1.5GB	230.7	410.6	580.3
3.3GB	210.6	264.5	326.4

Table 8: Server static workload capacities (Mb/s)

dicating that our latency benefits do not stem purely from extra capacity.

6.3 Excess parallelism

We also observe that all servers tested show latency degradation when running with more processes, though the effect is much lower for our new server. This observation is in line with the self-interference between the helpers and the main Flash process which we described earlier. We increase the number of helper processes and measure its effect on the SpecWeb99 results, as shown in Table 9. We observe that too few helpers is insufficient to fully utilize the disk, and increasing their number initially helps performance. However, the blocking from self-interference increases, eventually decreasing performance. A similar phenomenon, stemming from the same problem, is also observed with Apache. Using DeBox, we find that Apache with 150 processes, sleeps 3667 times per second, increasing to 3994 times per second at 300 processes. This behavior is responsible for Apache's latency increase in Figure 11.

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

Table 9: Parallelism benefits and self-interference – The conformance measurement indicates how many requests meet SpecWeb99's quality-of-service requirement.

This result suggests that excess parallelism, where server designers use parallelism for convenience, may actually degrade performance noticeably. This observation may explain the latency behavior reported for Haboob [41].

7 Results Portability

The main goal of this work is to provide developers with tools to diagnose and correct the performance problems in their own applications. Thus we hope that the optimizations made on one platform also have benefit on other platforms. To test this premise, we test the performance on Linux, which has no DeBox support.

Unfortunately, we were unable to get Linux to run properly on our existing hardware, despite several attempts to resolve the issue on the Linux kernel list. So, for these numbers, we use a server machine with a 3.0 GHz Pentium 4 processor and two Intel Pro1000/MT Gigabit adapters, 1GB of memory, and a similar disk. The experiments were performed on 2.4.21 kernel with `epoll()` support.

We compare the throughput and latency of four servers: Apache 1.3.27, Haboob, Flash, and the new Flash. We increase the max number of clients to 1024 in Apache and disable logging. Both the original Flash and the new Flash server use the maximum available cache size for LRU. We also adjust the cache size in Haboob for the best performance. The throughput results, shown in Table 10, are quite surprising. The Haboob server, despite having aggressive optimizations and event-driven stages, performs slightly better than Apache on diskbound workload but worse than Apache on an in-memory workload. We believe that its dependence on excess parallelism (via its threaded design) may have some impact on its performance. The new Flash server gains about 17-24% over the old one for the smaller workloads, and all four servers have similar throughput on the larger workload because of diskbound.

Throughput (Mb/s)				
data set	Haboob	Apache	Flash	New Flash
500MB	324.9	434.3	1098.1	1284.7
1.5GB	303.4	372.4	661.7	822.5
3.3GB	184.1	177.4	173.8	199.1

Response Time (ms)				
profile	Haboob	Apache	Flash	New Flash
5%	78.2	0.22	0.21	0.15
median	414.3	0.61	1.56	0.42
95%	1918.9	661.8	412.5	3.68
mean	656.2	418.0	512.5	141.9

Table 10: Throughput measurement on Linux with 1GB memory

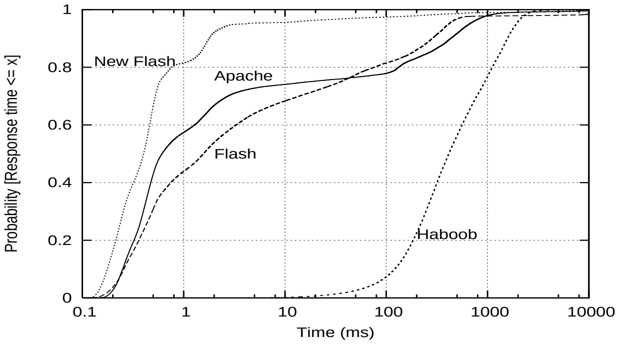

Figure 12: Response time on Linux with 3.3GB dataset

Despite similar throughputs at the 3.3GB data set size, the latencies of the servers, shown in Figure 12 and Table 10, are markedly different. The Haboob latency profile is very close to their published results, but are beaten by all of the other servers. We surmise that the minimal amount of tuning done to configurations of Apache and the original Flash yield much better results than the original Haboob comparison [41]. The benefit of our optimization is still valid on this platform, with a factor of 4 both in median and mean latency over the original Flash. One interesting observation is that the 95% latency of the new Flash is a factor of 39 lower than the mean value. This result suggests that the small fraction of long-latency requests is the major contribution to the mean latency. Though our Linux results are not directly comparable to our FreeBSD ones due to the hardware differences, we do notice this phenomenon is less obvious on FreeBSD. Presumably one of the causes of this is the blocking disk I/O feature of `sendfile()` on Linux. Another reason may be Linux's filesystem performance, since this throughput is worse than what we observed on FreeBSD.

8 Related Work

To compare DeBox's approach of making performance information a first-class result, we describe three categories of tools currently in use, and explain how DeBox relates to these approaches.

- **Function-based profilers** – Programs such as `prof`, `gprof` [18], and their variants are often used to detect hotspots in programs and kernels. These tools use compiler assistance to add bookkeeping information (count and time). Data is gathered while running and analyzed offline to reveal function call counts and CPU usage, often along edges in the call graph.
- **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 high-precision information at lower cost. The other major difference is that these profilers, such as DCPI [4], Morph [43], VTune [19], Oprofile [29], and PP[3] tend to be whole system profilers, capturing activity across all processes and the operating system.

In this category, DeBox is logically closest to kernel `gprof`, though it provides more than just timing information. DeBox's full call trace allows more complete call graph generation than gprof's arc counts, and with the data compression and storage performed in user space, overhead is moved from the kernel to the process. Compared to path profiling, DeBox allows developers to customize the level of detail they want about specific paths, and it allows them to act on that information as it is generated. In comparison to low-level statistical profilers such as DCPI, coverage differs since DeBox measures functions directly used during the system call. As a result, the difference in approach yields some differences in what can be gathered and the difficulty in doing so – DCPI can gather bottom-half information, which DeBox currently cannot. However, DeBox can easily isolate problematic paths and their call sites, which DCPI's aggregation makes more difficult.

- **System activity monitors** – Tools such as `top`, `vmstat`, `netstat`, `iostat`, and `systat` can be used to monitor a running system or determine a first-order cause for system slowdowns. The level of precision 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 access to source code. Tools such as `truss`, PCT [11], `strace` [2], and `ktrace` are able to show some details of system calls, such as parameters, return values, and timing information. Recent tools, such as Kitrace [21] and the Linux Trace Toolkit [42], also provide data on some kernel state that changes as a result of the system calls. These tools are intended for observing another process, and as a result, producing out-of-band measurements and data aggregation, often requiring post-processing to generate usable output.
- **Timing calls** – Using `gettimeofday()` or similar calls, programmers can manually record the start and end times of events to infer information based on the difference. The `getrusage()` call adds some information beyond timings (context switches, faults, messages and I/O counts) and can similarly used. If per-call information is required, not only do these approaches introduce many more system calls, but the information can be misleading.

DeBox compares favorably with a hypothetical merger of the timing calls and the trace tools in the sense that timing information is presented in-band, but so is the other information. In comparison with the Linux Trace Toolkit, our focus differs in that we gather the most significant pieces of data related to performance, and we capture it at a much higher level of detail.

- **Microbenchmarks** – Tools such as lmbench [24] and hbench:OS [13] can measure best-case times or the isolated cost of certain operations (cache misses, context switches, etc.). 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 [33] helps automate the process of pinpointing latency. The work of Cota-Robles and Held [16] and Jones and Regehr [20] 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 [40], ParaDyn [25], Etch [35], and ATOM [37].

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. One major difference between DeBox and kernel instrumentation is that we provide a rich set of measurements to any process, rather than providing information only to privileged processes.

Beyond these performance analysis tools, 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 the Infokernel system by Arpaci-Dusseau et al. [8]. Our goals differ, since we are more concerned with understanding why blocking occurs rather than reacting to it during a system call. Our non-blocking `sendfile()` modification 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 aborts if the necessary metadata is not in memory [26].

Our observations on blocking and its impact on latency may impact server design. Event-driven designs for network servers have been a popular approach since the performance studies of the Harvest Cache [12] and the Flash server [30]. Schmidt and Hu [36] performed much of the early work in studying threaded architectures for improving server performance. A hybrid architecture was used by Welsh et al. [41] to support scheduling, while Larus and Parkes [22] demonstrate that such scheduling can also be performed in event-driven architectures. Qie et al. [34] 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.

9 Conclusions and Discussion

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 and user interactions correlated with user-level events. Furthermore, we believe that the ability to monitor behavior on-line provides programmatic flexibility 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.

We have shown how DeBox can be used in a variety of examples, allowing developers to shape profiling policy and react to anomalies in ways that are not possible with other tools. Although DeBox does require access to kernel source code for achieving the highest impact, we do not believe that such a restriction is significant. FreeBSD, NetBSD, and Linux sources are easily available, and with the advent of Microsoft's Shared Source initiatives, few hardware platforms exist for which some OS source is not available. Also, general information about kernel behavior instead of source code may be enough to help application redesign. Our performance portability results also demonstrate that our new system achieves better performance even without kernel modification. A further implication of this is that it is possible to perform analysis and modifications while running on one operating system, and still achieve some degree of benefit in other environments.

In this paper we focused on how DeBox can be used as a performance analysis tool, but we have not discussed its utility in general-purpose monitoring because of space limits. Given its low overheads, DeBox is an excellent candidate for monitoring long-running applications. We are approaching this problem by modifying the `libc` library and associated header files so that a simple recompile and relink will enable monitoring of applications using DeBox. It is also possible to process results automatically by allowing user-specified analysis policies. We are working on such a tool, which will allow passive monitoring of daemons, but a full discussion of it is beyond the scope of this paper.

While we have shown DeBox to be effective in identifying performance problems in the interaction between the OS and applications, the current version of DeBox does not handle the bottom-half activities in the kernel. DeBox's current focus on the system call boundary also makes it less useful for tracing problems arising purely in user space. However, we believe that both of these limitations can be addressed, and we are continuing work in these areas.

Acknowledgments

This work was partially supported by an NSF CAREER Award. We would like to thank our shepherd, Remzi Arpaci-Dusseau, and our anonymous reviewers for their feedback and insight. We would also like to thank the FreeBSD developers, particularly Mike Silbersack and Alan Cox, for their efforts in integrating our `sendfile()` modifications.

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. 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] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proc. of the 18th ACM Symp. on Operating System Principles*, pages 90–105, Bolton Landing, NY, Oct. 2003.

[9] 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.

[10] 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.

[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 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.

[18] 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.

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

[20] 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.

[21] G. Kuenning. Kitrace—precise interactive measurement of operating systems kernels. *SOFTWARE–PRACTICE AND EXPERIENCE*, 1(1):1–21, 1994.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] OProfile. http://oprofile.sourceforge.net/.

[30] 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.

[31] 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.

[32] 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.

[33] 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*, San Jose, CA, Sept. 2002.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.