Poking Holes in the Black Box: An Investigation of Run-Time Kernel Performance Improvements

William C. Benton
Brian L. Bowers
Keith D. Noto
{willb,blbowers,noto}@cs.wisc.edu

May 2, 2002

Abstract

General-purpose operating systems expose an interface to hardware resources such as memory, disk, and network devices, and mediate access to the CPU. Well-designed general-purpose operating systems export resources as instances of a fundamental abstractions with a set of associated operations and well-defined semantics, such as the "everything is a file" concept in UNIX. Operating system abstractions make life easier for application developers and allow transparent access to a variety of resources, but can be inefficient and limiting. We examine a variety of techniques which break the fundamental abstraction of an operating system to allow adapability, extensibility, and high-performance, and evaluate the performance improvements that they afford in several application domains.

1 Introduction

Since many applications are built upon a palimpsest of libraries, system services, and cooperating processes, it is often difficult for users to see where their performance problems are coming from. The performance of the most efficiently written user-level program can be adversely affected by poorly-written or poorly-specialized libraries, software components, or system services. This problem of layered inefficiencies is exacerbated by the fact that good software engineering practice encourages developers to regard the implementation of an interface as a black box.

There are many ways that libraries and kernels can make well-meaning software inefficient, including

- system calls that compute dead or redundant values or that are redundant in their effects

- system calls that make unnecessary or unintended semantic guarantees
- unnecessary copying between user space and kernel space
- excessive context switch time
- hiding any of the above beneath several layers of innocent-seeming interface

In this work, we aim to explore and expose the sources of some of these insidious software inefficiencies, to explore the sorts of performance improvements that are possible if one is willing to break interface boundaries, and to suggest a method for enabling these improvements dynamically.

2 Contributions of this research

We present an overview of foundational and current research into extensible, adaptive, and evolving operating systems in section 3. This presents the reader with the state of the art in the field and provides background for the problems we faced and the solutions we chose.

In section 4, we investigate the performance of representative applications from several domains:

- web proxy applications
- database servers
- graphical end-user productivity applications

Specifically, we examine the performance of the `squid` web proxy [**?**], the PostgreSQL database server [**?**], the `xpdf` document viewer [**?**], and the popular Mozilla web browser [**?**]. We then discuss our evaluation of high-performance audio applications under Linux, which are not as amenable to our performance improvements as the other applications we examined.

We then, in section 5, evaluate the benefits of various improvement techniques which cross the application/kernel boundary; these include:

- "macro-system calls", or pushing application logic into the kernel
- specialized versions of system calls
- "semantics-driven cooperation" between application and kernel, or replacing one sequence of calls with an equivalent, less expensive one

We base our evaluation on performance increases gained by applying these techniques statically to representatives several application

Note that we deliberately do not develop a full-blown extension system for the kernel; we agree with Peter Druschel [DPZ97] that, while extensible operating systems introduce many interesting problems to solve, that time spent solving those problems does not solve the problems of operating systems that seem to necessitate extensible systems in the first place.

We discuss our plans for future work in section 6 and conclude in section 7.

3 Related work

Systems and language researchers have, in the last few years, developed a number of orthogonal approaches to building adaptive, extensible, and high-performance operating systems, as well as a number of orthogonal approaches to solving some of the problems presented by modifying an operating system's behavior at run time. In this section, we present a taxonomy of the approaches developed by various researchers, the constituent problems entailed by each approach, and the solutions that various groups developed to solve those problems.

We examine adaptive, extensible, and high-performance systems together because we see them as partners in a symbiotic relationship: an extensible system can be used to prototype or simulate an adaptive system; a well-designed adaptive system in turn obviates some (but not all) of the necessity of an extensible system. High-performance systems are relevant to our discussion because most performance improvements[1] make nontrivial assumptions about application behavior – improvements which benefit applications from one application domain might hinder those from another. Furthermore, adding extra complexity to a kernel for a special case will likely hurt performance in the general case, and vice versa. As a result, it seems beneficial to enable application-driven performance improvements.

3.1 What kind of system?

The strongest distinction that we can initially draw is between projects that aim to improve a commodity kernel and those that aim to improve a research kernel. This also introduces a key conflict in operating systems research – the problem of POSIX/UNIX compatibility – which we

[1]Throughout this work, we use the expression "performance improvements" instead of the more common "optimizations" because we are unwilling to imply that the result of several performance improvements is a system with optimal performance.

Based on commodity systems	Based on research systems
IO-Lite	Exokernel
KernInst	SPIN
Synthetix	
VINO	

Figure 1: Some extensible and high-performance operating systems research projects, categorized by whether they modify a commodity system or not.

will discuss shortly. Projects which operate on commodity systems have the benefit of running "real applications" such as `gcc`, `emacs`, and the Apache web server; on the other hand, projects which develop their own kernels are free from the shackles imposed by compatibility with an existing system and its constituent problems: preëxisting, inflexible, general-purpose and inefficient mechanisms and unnecessary interfaces. Figure 1 presents a classification of some existing systems into these two categories.

We now examine the approaches of these systems briefly, beginning with those which modified commodity systems:

- *IO-Lite* [PDZ00] was developed by Vivek Pai, Peter Druschel, and Willy Zwanenpoel at Rice University. IO-Lite is a high-performance extension to BSD to allow zero-copying I/O with immutable I/O buffers. Since much of the time spent servicing I/O requests in a traditional UNIX system involves copying data from user space to kernel space and vice versa, IO-Lite presented impressive performance gains. IO-Lite exposes zero-copying I/O to application developers via special, nonstandard system calls. IO-Lite effectively offers users an end-run around the semantics of I/O buffers provided by standard UNIX systems.

- *KernInst* [TM99a], developed by Ari Tamches and Bart Miller at Wisconsin, extends the technique of dynamic program instrumentation [HMC94] to the Solaris kernel, allowing the insertion of runtime-generated code at arbitrary points in the kernel. Tamches uses dynamic instrumentation to tune applications which make incorrect assumptions about the performance characteristics of Solaris [TM99b] and to improve the instruction-cache behavior of hot routines in the Solaris kernel [Tam01].

- *Synthetix* [PAB+95], developed by a team lead by Calton Pu, integrates a partial evaluator into the HP-UX kernel, to improve system performance by producing specialized versions of system calls. Partial evaluation has seen great effect in improving the performance

of ray-tracing systems [And96] and in effectively building specialized software components [Vel99, CDPR98]. Those problem domains share a common concern with operating system kernels: taking a general case (a ray-tracing function or a template for a routine) and reducing it to a special case (a function which traces a particular scene or a specialized routine) while improving performance. Operating system kernels provide a general interface which is unnecessarily complicated for most user-space applications; as a result, a wide range of applications are possible, but every application pays a performance penalty for every special case in every operating system service that it uses.

- *VINO* [SS94], developed by Christopher Small and Margo Seltzer at Harvard, presents an extensible, preemptible UNIX-like kernel with fine-grained locking. VINO is based to some degree on NetBSD, in that most of its device drivers and user-level tools are derived from NetBSD code, and it is mostly UNIX-compatible. However, the core of VINO supports uploading sandboxed extensions into the kernel, and VINO is also capable of self-modifying and self-adapting to different workload types.

The projects that improve commodity systems demonstrate major improvements for real-world applications. However, a committment to UNIX is, in many ways, an unnecessary set of shackles for OS research. Several researchers, including Rob Pike [Pik00], have railed against the fact that UNIX-centric projects still dominate the OS research field while arguably better ideas like microkernels and capability-based systems have fallen into disuse. While this is true for OS research in general, adaptive and extensible systems researchers have developed many non-UNIX-based projects, since the unique requirements of dynamic extensibility and improvement are often in conflict with the large monolithic designs of most UNIX systems. We now examine the projects based on research OS designs.

- *Exokernel* [EKO95], developed by Dawson Engler and others from Frans Kaashoek's PDOS lab at MIT, takes the end-to-end approach [SRC84] to kernel design. The fundamental assumption of Exokernel is that operating system abstractions of resources are irrelevant, inflexible and wasteful; instead, the operating system should simply provide a way to allow applications to concurrently access hardware safely. In this respect, the assumptions of the Exokernel system are similar in spirit to the assumptions of the Synthetix system – that operating systems attempt to provide general coarse functionality and application-specific guarantees, instead of providing the building blocks for applications to make their own system

Trusts extensions	Does not trust extensions	Does not have to trust extensions
Linux	KernInst	VINO
SPIN	PCC	Exokernel

Figure 2: Some operating systems, categorized by their trust model for extension code.

functionality with application-specific semantics and guarantees. Of course, the approaches each takes are quite different: Synthetix produces system calls with specialized semantics and performance characteristics for different applications, while Exokernel provides a RISC-like substrate for building meaningful coarse-grained system services at user level. Exokernel actually has a functional UNIX compatibility layer at user level, which exists as a shared library and can coexist with other higher-level OS functionality layers.

- *SPIN* [BCE+94], developed at the University of Washington, is a research system written in Modula-3, a typesafe, high-performance language. Since a program developed in Modula-3 will have certain "good-behavior" properties – for example, it will have no buffer overflows, misaligned reads, or pointer arithmetic – SPIN allows arbitrary users to upload arbitrary extensions, written in Modula-3, to the kernel. SPIN applications, then, can run entirely in user-space, entirely in kernel-space, or in some combination of both. SPIN also features a Digital UNIX compatibility layer in user-space.

3.2 What kind of safety?

The major issue we examine is that of code safety. Systems that allow applications to dictate or modify kernel policy or semantics *must* provide some means for guaranteeing that the changes are safe, since a rogue extension which crashes a kernel will result in a crashed system. Various systems take radically different approaches to this problem, which we categorize by what we see as three different fundamental assumptions: "I trust extensions", "I don't trust extensions", and "I don't *have to* trust extensions". We summarize our categorization of these systems in Figure 2, and examine each trust category in turn.

3.2.1 "I trust extensions"

Linux, like many modern UNIX systems, supports adding behavior to the kernel at runtime via loading modules which can define additional functions in kernel space, define callbacks from kernel events, and modify exported kernel symbols. This is the standard mechanism for adding device

drivers to a running system, but has also been used to modify kernel policy [BD01, pp99] at runtime. Because Linux modules are unsigned, unsafe assembly code (typically generated from a C compiler), run in the kernel's address space, and can modify kernel data, a faulty module can crash the whole system. In this way, Linux is an example of why blindly trusting extensions is a recipe for disaster.

SPIN, on the other hand, uses the safety guarantees provided by the Modula-3 language to ensure that uploaded code cannot crash the kernel. In this way, it is analogous to the protection model provided by the Pilot operating system [RDH+79], in which memory protection was obviated by the fact that all user processes were implemented in the safe Mesa language. Obviously, a mechanism which relies on language-level protections can be subverted simply by subverting the language, but if a system administrator trusts the users of her system, language-based security is adequate to ensure that well-meaning but faulty extensions cannot crash the system.

3.2.2 "I don't trust extensions"

KernInst inserts machine code snippets into a running kernel. Since these snippets are generated by the programmer (and since developing machine code is error-prone), they are by default untrusted. Since these snippets are often quite small, their safety properties are statically analyzed before insertion, using a method developed by Zhichen Xu [XMR00]. Static analyses of machine code are quite expensive, so they are not suitable for whole-programs, but they are well-suited to snippets that collect profile data or replace small functions.

An alternative solution to untrusted extensions is presented by George Necula and Peter Lee in their implementation of proof-carrying code for kernel extensions [NL96]. Proof-carrying code combines a program with an automatically-generated proof of its safety, and shares a major advantage with static analysis – because programs are verified before they are loaded, they have no opportunity to put the system in an inconsistent state via a software fault. Furthermore, verifying a proof is nontrivial, but is generally less expensive than proving properties about an unknown binary. However, the major disadvantage of proof-carrying code is that it moves the boundary of trust from the program which does the verification (the kernel) to the program which produces the code (the compiler). Since there is no way to verify that a proof corresponds to a program without doing an expensive (and possibly intractable) analysis of the program itself, to trust a program based on its proof is also to trust the compiler that produces the code and proof – in this sense,

proof-carrying code is similar to language-based protection mechanisms.

3.2.3 "I don't *have to* trust extensions"

We now examine two approaches which bypass the issue of whether or not to trust code; both accomplish their aims by isolating the fault domain of the extension code. In this way, these approaches are analogous to memory protection in modern operating systems: a rogue process cannot crash another process. Likewise, isolation techniques aim to prevent an operating system extension from damaging the rest of the system if it crashes. Just as an operating system makes no attempt to verify safety properties of a program before running it, fault-isolated systems have no need to verify extensions before loading them.

The VINO system loads "grafts", or C++ extensions, into its address space at run-time. However, it sandboxes the code into its own isolated fault domain, so that it cannot alter other kernel code. This approach allows untrusted code to run safely in the same address space as trusted code, but at the cost of a performance penalty for loads and stores, to ensure that untrusted code does not modify code or data in a different fault domain.

Exokernel, on the other hand, provides only the most basic hardware access as operating system primitives; therefore, most of the services that are traditionally considered operating system services run in user space as "library operating systems", so they are as isolated as seperate user processes in a traditional operating system. In fact, if a process installs its own library operating system on top of the low-level Exokernel substrate, no other software failures will affect it.

4 Performance studies

In this section, we briefly introduce the applications we are benchmarking, discuss the performance characteristics we observed in these applications and begin to discuss the approaches we take to improve their performance.

4.1 The PostgreSQL database

PostgreSQL is an advanced object-relational database management system that has as its origins the Ingres, Postgres and Postgres95 projects at Berkeley. PostgreSQL is the most advanced database

that is distributed as Free[2] software (under the BSD license), and is widely deployed in real-world scenarios. Therefore, it is an ideal candidate for our tests.

Initial traces indicate that PostgreSQL spends a great deal of time on disk activity, network activity over UNIX and TCP/IP domain sockets, and shared memory access.

Note to reviewers: Until recently (May 1), we were unable to collect profiling data for multiprocess applications. We have since discovered an adequate multiprocess profiler, `vprof`. Expect performance analyses at least as detailed as the squid analysis in the final paper.

4.2 The `squid` proxy

Squid is a web proxy. As part of its operation, Squid must receive and cache numerous web pages, saving them to disk for later reuse. The nature of a proxy application means that the files are all transient, are accessed frequently, and are overwritten when evicted from the cache. Using `strace` on Squid, we saw many occurrences of open-write-close system calls, providing evidence for this intuition.

Our efforts to determine the amount of time spent using open-write-close were blocked by the lack of a good multi-process profiler. Squid starts a main process then spawns a child process to handle much of the real work.

We proceeded by generating synthetic benchmarks that use the same sequence of calls. Our baseline code was take verbatim from Squid. With this baseline code, we generated some initial costs of using open-write-close, an alternative open-mmap-memcpy-munmap-ftruncate-close, and finally a smarter variation on open-write-close.

The current method for Squid uses the code below:

```
fd = open(name, O_TRUNC|O_WRONLY|O_NONBLOCK);
write(fd, buffer, size);
close(fd);
```

We tested the Squid code against an mmap version, shown below:

```
fd = open(name, O_RDWR|O_NONBLOCK, 0644);
memPtr = mmap(  0,
                size,
                PROT_READ|PROT_WRITE,
```

[2]We capitalize "free" to indicate that we refer to freedom, not monetary cost – for our purposes, the freedoms to examine and modify the source code and to publish benchmarks are particularly significant.

% time	inclusive seconds	exclusive seconds	calls	inclusive seconds/call	exclusive seconds/call	function name
74.42	542.86	542.86	1	542856.00	542856.00	openWriteCloseTest
25.58	729.46	186.60	1	186604.00	186604.00	mmapTest
0.00	729.47	0.01	1	12.00	729472.00	main

Table 1: The first comparison of open-write-close and mmap

```
                    MAP_PRIVATE,
                    fd,
                    0);
memcpy(memPtr, buff, size);
munmap(memPtr, size);
close(fd);
```

Our testbed program contained two methods, one using the open-write-close sequence and one using the open-mmap-memcpy-munmap-close sequence. The methods actually performed 10000 iterations of the sequences to get an aggregate time. The results, as drawn from hrprof, are shown in table 1.

Saving $\frac{2}{3}$ of the time for these sequences of calls seems promising. Unfortunately, a problem arises when the amount of data to be written is larger than the current file size. Linux does not let you `mmap()` more space than is available in the file; writing past the end of the file results in a segmentation fault. We had to find a better way to get the job done.

We looked at three possible cases: the existing file is at least as large as the data to be written (our method above works); the existing file is smaller than the data to be written; the file has no space (a new file). We came up with a simple set of tests that would work correctly, as shown below:

```
fd = open(name, O_CREAT|O_NONBLOCK|O_RDWR, 0644);
fstat(fd, &buf); /* buf is a struct stat object */
if (buf.st_size >= size) {
        memPtr = mmap(  0,
                        size,
                        PROT_READ|PROT_WRITE,
                        MAP_PRIVATE,
                        fd,
                        0);
        memcpy(memPtr, buffer, size);
        munmap(memPtr, size);
        ftruncate(fd, size);
} else if (buf.st_size > 0) {
        memPtr = mmap(  0,
                        buf.st_size,
```

% time	inclusive seconds	exclusive seconds	calls	inclusive seconds/call	exclusive seconds/call	function name
70.04	802.44	802.44	1	802.44	802.44	writeTest
29.95	1145.59	343.15	1	343.15	343.15	mmapTest
0.00	1145.61	0.02	1	0.02	1145.61	main

Table 2: Wall times for the modified testbed

```
                        PROT_READ|PROT_WRITE,
                        MAP_PRIVATE,
                        fd,
                        0);
        memcpy(memPtr, buffer, buf.st_size);
        munmap(memPtr, buf.st_size);
        lseek(fd, 0, SEEK_END);
        write(fd, buffer + buf.st_size, size - buf.st_size);
} else {
        write(fd, buffer, size);
}
```

Our modified testbed provided the run times shown in table 2,

which represent a significant time savings. But if `mmapTest` was using `write()` in some parts of the execution, why didn't it suffer as badly as `writeTest`? We decided that there must be a hidden cost.

Using `O_TRUNC` during `open()` forced the system to release any backing store that might have been allocated for the file. When trying to write, even a small amount of data, more backing store was allocated. Getting backing store would almost certainly be a disk operation, with the inherent costs. Our test was comparing different costs - perhaps `write()` was the culprit, but we couldn't be sure.

We changed our setup to allow the normal write to use existing backing store when possible. We removed the `O_TRUNC` flag from the call to `open()` and added an `ftruncate()` when necessary at the end of the original `write`, giving the code below:

```
fd = open(name, O_NONBLOCK|O_WRONLY, 0644);
write(fd, buffer, size);
ftruncate(fd, size);
close(fd);
```

Our results here, shown in table 3 were comparable to using `mmap()`, although `mmap()` still required 8% to 10% less time than `write()`. The call to `open()`, thus, was the cause of the observed inefficiency. This is consistent with the results observed by Tamches [TM99b] in his study of squid.

% time	inclusive seconds	exclusive seconds	calls	inclusive seconds/call	exclusive seconds/call	function name
52.12	396.32	396.32	1	396325.00	396325.00	writeTest
47.88	760.39	364.06	1	364063.00	364063.00	mmapTest
0.00	760.40	0.02	1	17.00	760405.00	main

Table 3: Results demonstrating the similar performance of `write` and `mmap` for a non-truncating `open`

4.3 The Mozilla web browser

The results of our investigation into the Mozilla web browser (version 0.9.2[3]) have been inconclusive. We have examined a number of statistics from this program, including output from strace, gprof, and hrprof (high-resolution profiling), but we have not yet identified any parts of the program made inefficient through the use of system calls. It remains to be seen whether or not any of our techniques will be applicable to this program as we continue our experiments.

Mozilla does have some curious statistics. Notably, 26.79% percent of time in a sample run was spent on two calls to `nsID::Equals`, the equality operator for a class.

We continue to investigate these results, and anticipate that the discovery of the advanced profiler `vprof` will aid us to this end.

4.4 The `xpdf` document viewer

`xpdf` is display-bound – over 30 percent of its time is spent communicating with the X server. We believe that there is room for improvement, as a comparable program, Adobe's `acroread`, is significantly faster.

We will quantify and discuss this further in the final paper.

4.5 Some high-performance audio applications

We investigated the performance of two high-performance digital audio workstation (DAW) applications: Ardour and ecasound. We had initially intended to also investigate the performance of several software synthesis applications, but the overwhelming evidence is that those are unsuitable for our performance improvements for two reasons:

[3]We deliberately choose a somewhat older version of Mozilla (the current version as of this writing is 1.0) in order to have a relatively unoptimized application to discover performance enhancement opportunities in.

- To the best of our knowledge, all available synthesis applications deal in 16- or 32-bit stereo samples at a frequency of between 22.05 khz and 96 khz. As a result, none are data-bound.
- Furthermore, because of the intensive computational power required for software synthesis, software synthesizers are largely CPU-bound.

Unfortunately, the performance characteristics of the audio applications that we examined indicate that they would not benefit from the sorts of improvements that we are investigating. The main reasons for this are:

- The high-performance Linux sound drivers, available at http://www.alsa-project.org (ALSA are the only drivers that support professional-grade multichannel audio hardware) enable zero-copying audio I/O.
- The popular higher-level Linux audio API (http://jackit.sf.net) is callback-based; as a result, audio is produced and consumed synchronously. As a result, there are no queueing effects or buffering issues.

We communicated with Ardour's author, who claimed that kernel scheduling latency is a serious issue for Linux audio; we were able to independently verify this assertion. However, implementing a configurable scheduling policy for Linux is outside of the capabilities of our techniques as well as outside of the scope of a single-semester project. Furthermore, there is no evidence that the projects to provide low-latency scheduling and a preemptible kernel (**we need citations here**), which improve the performance characteristics of high-performance real-time applications, interfere with the performance of ordinary workstation computing. We are interested in investigating this further (at least one of us is quite interested in audio), but do not feel that we can currently contribute in this area.

5 Performance improvement evaluation

To be completed – we have a library interposer for squid that enables relaxed file semantics with the current file access interface. We also have concrete solutions for improving PostgreSQL, xpdf, and Mozilla, and are furiously implementing and measuring them.

5.1 Improvements to `squid`

5.2 Improvements to Mozilla

5.3 Improvements to `xpdf`

5.4 Improvements to PostgreSQL

6 Future work

To be completed

7 Conclusion

To be completed

References

[AFG+97] Marc A. Auslander, Hubertus Franke, Benjamin Gamsa, Orran Krieger, and Michael Stumm. Customization lite. In *Workshop on Hot Topics in Operating Systems*, pages 43–48, 1997.

[And96] Peter Holst Andersen. Partial evaluation applied to ray tracing. In W. Mackens and S.M. Rump, editors, *Software Engineering im Scientific Computing*, pages 78–85. Vieweg, 1996.

[BCE+94] Brian N. Bershad, Craig Chambers, Susan J. Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN - an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop*, pages 68–71, 1994.

[BD01] William C. Benton and Tao Di. An overview of FIDELIO, the Facility for Indetectable Dynamic Event Logging and Intrusion Observance. 2001.

[CDPR98] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic programming by program composition (position paper). In *Workshop on Generic Programming*, Marstrand, Sweden, 1998.

[DPZ97] Peter Druschel, Vivek S. Pai, and Willy Zwaenepoel. Extensible systems are leading OS research astray. In *Workshop on Hot Topics in Operating Systems*, pages 38–42, 1997.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[FP93] Kevin R. Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve i/o throughput and CPU availability. In *USENIX Winter*, pages 327–334, 1993.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, University of Wisconsin, 1994.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[PAB+95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO (USA), 1995.

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, 1999.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[Pik00] Rob Pike. Systems software research is irrelevant. Presentation slides available from www.cs.bell-labs.com/who/rob/, 2000.

[pp99] "pragmatic/THC" (pseud.). (Nearly) Complete Linux Loadable Kernel Modules. Web published at http://packetstorm.securify.com/docs/hack/LKM_HACKING.html, March 1999.

[RDH+79] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An operating system for a personal computer. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 106–107, 1979.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Operating Systems Design and Implementation*, pages 213–227, 1996.

[SESS97] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Issues in extensible operating systems. Technical Report TR-18-97, Harvard University, 1997.

[Sma98] Christopher Small. *Building an Extensible Operating System.* PhD thesis, Harvard University, Cambridge, MA, 1998.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SS94] Christopher Small and Margo I. Seltzer. VINO: An integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard University, Cambridge, MA, 1994.

[SS96] Christopher Small and Margo I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.

[Tam01] Ariel Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating Systems.* PhD thesis, University of Wisconsin, 2001.

[TM99a] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[TM99b] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3):263–276, Fall 1999.

[Vel99] Todd L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18, 1999.

[WBvE97] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. Technical Report TR97-1620, Cornell University, 2, 1997.

[XMR00] Zhichen Xu, Barton P. Miller, and Thomas W. Reps. Safety checking of machine code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–82, 2000.