Cosy: Develop in User-Land, Run in Kernel-Mode

Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok
Stony Brook University

Appears in the 9th Workshop on Hot Topics in Operating Systems (HOTOS 2003)

Abstract

User applications that move a lot of data across the user-kernel boundary suffer from a serious performance penalty. We provide a framework, Compound System Calls (Cosy), to enhance the performance of such user-level applications. Cosy provides a user-friendly mechanism to execute the data-intensive code segment of the application in the kernel. This is achieved by aggregating the data-intensive system calls and the intermediate code into a compound. This compound is executed in the kernel, avoiding redundant data copies.

A Cosy version of GCC makes the formation of a Cosy compound simple. Cosy-GCC automatically converts user-defined code segments into compounds. To ensure the security of the kernel, we use a combination of static and dynamic checks. We limit the execution time of the application in the kernel by using a modified preemptible kernel. Kernel data integrity is assured by performing necessary dynamic checks. Static checks are enforced by Cosy-GCC. To study the performance benefits of our Cosy prototype, we instrumented applications such as `grep` and `ls`. These application showed an improvement of 20–80%. Our current work focuses on faster and secure execution of entire programs in the kernel without source code modification.

1 Introduction

User applications like HTTP, FTP, and mail servers involve copying a significant amount of data across the user-kernel boundary. It is well understood that this cross-boundary data movement is expensive. It can reduce the overall performance of the application by two orders of magnitude [11]. For applications like `ls` that invoke a large number of small system calls, performance is hampered by the time wasted in context switching.

A solution to improve performance of such applications is to move the bottleneck code segment, the segment involving cross-boundary data movement, into the kernel [1, 5]. Applications written for VINO [5] and SPIN [1] have supported this idea. Exokernels [3] take a different approach by allowing user processes to describe the on-disk data structures and methods to implement them. The common problem with these approaches is that they do not fit in the framework of current commodity operating systems. Others attempted to improve the performance but at the cost of safety [8]. Software fault isolation provides a secure mechanism to execute untrusted code, but performance benefits are often lost [9]. Other solutions to improve performance include `sendfile`, NFSv4 [6], and smart storage [7]. Sendfile provides a way to avoid data copies for network-specific applications. NFSv3 adds some new calls like READDIRPLUS to reduce the overhead due to a `readdir` followed by several `stat` calls. Smart storage is limited to improving disk I/O and can not be extended to networks. These approaches are successful but they are not extensible.

We present Cosy as a generic solution to safely execute bottleneck code segments of user applications in the kernel. Cosy exploits zero-copy techniques and code aggregation to achieve better performance without reducing the security. Cosy extracts the system calls and the intermediate code from the bottleneck code segment and encodes them to create a *compound*. A *compound* is formed by aggregation of system calls, programmatic constructs (e.g., `if-else` and `while`), and user specified functions. This compound is then passed to the kernel via a new system call (`cosy_run`) which decodes it and executes the decoded elements in the compound intelligently to avoid redundant data copies.

To facilitate automatic generation of compounds we provide Cosy-GCC, a modified version of GCC, which makes writing Cosy compounds simple. The user just needs to mark the bottleneck region and Cosy-GCC converts it into a compound at compile time. Cosy-GCC also finds data dependencies among Cosy statements and encodes this information into the compound. This information is used by the kernel while executing the compound to avoid data copies.

Systems that allow arbitrary user code to execute in kernel mode must address security and protection issues: how to avoid buggy or malicious code from corrupting data, accessing protected data, or crashing the kernel. Securing such code often requires costly runtime checking [8]. Cosy uses a combination of static and dynamic approaches to assure safety in the kernel. Cosy explores various hardware features along with software techniques to achieve maximum safety without adding much overhead.

The rest of this paper is organized as follows. Section 2 describes the design of our system. Section 3 presents an evaluation of our Cosy prototype. We conclude in Section 4.

2 Design

The main motivation behind Cosy is to achieve maximum performance with minimal user intervention, and without compromising security. The primary design goals were as follows:

Performance We exploit several zero-copy techniques at various stages to enhance the performance. We make use of shared buffers between user and kernel space for fast cross-boundary data exchange.

Safety We make use of various security features including kernel preemption and x86 segmentation to assure a robust safety mechanism. We make use of a combination of static and dynamic checks to assure safety in the kernel without adding run-time overheads.

Simplicity We have automated the formation and execution of the compound so that it is almost transparent to the end user. Thus it is simple to write new code as well as modify existing code to use Cosy. The Cosy framework is extensible and adding new features to it is not difficult.

2.1 Architecture

Cosy executes compounds of system calls in the kernel. Often only small sections of code suffer from cross-boundary communication. The first step while using Cosy is to identify these *bottleneck* code segments. A bottleneck code segment is transformed into a compound by identifying and aggregating the system calls, arithmetic, assignment operations, loops, and function calls. To facilitate the formation and execution of a compound, Cosy provides three components: the Cosy kernel extension, Cosy-GCC, and Cosy-Lib. These components communicate using two shared buffers. The *compound buffer* is a shared buffer used to encode the compound. The *fast buffer* is another buffer used to facilitate zero-copy within system calls that share arguments. We look at the individual components and the internals of the compound buffer in the following subsections.

2.1.1 Kernel Extension for Cosy

The Cosy kernel extension exposes three system calls to the user space components.

- **cosy_init** allocates the two shared buffers that are used by the user and kernel to exchange the encoded compound and the results.
- **cosy_run** decodes the compound from the compound buffer and executes the decoded instructions one by one.
- **cosy_uninit** frees any Cosy resources for the current process including the shared buffers.

Each Cosy-enabled process has its own shared buffers.

2.1.2 Cosy-GCC

Currently, the application programmer just needs to identify the bottleneck code segment in the application and mark it in the standard C program using the markers provided by Cosy (COSY_START and COSY_END). Usually, no modifications are needed to the code within these markers. The only constraint is that all the instructions within the marked block should be supported under Cosy-GCC. Cosy-GCC parses the code and if all the instructions within the segment are supported then it modifies the marked code. It also inserts a `cosy_run` at the end of the marked segment. The code is modified in such a way that during execution, the modified code forms a compound and the `cosy_run` at the end informs the Cosy kernel extension to execute this recently-formed compound. Cosy-GCC also maintains a symbol table of labels for Cosy calls. This symbol table is used to find out any interdependency among the arguments of compounded calls. The information about interdependencies is also encoded in the compound buffer. The Cosy kernel extension uses this information to avoid data copies. The symbol table is also used to resolve the jump labels.

Cosy supports loops (e.g., `for`, `do-while`, and `while`), conditional statements (e.g., `if`, `switch`, and `goto`), simple arithmetic operations (e.g., increment, decrement, assignment, add, and subtract) and system calls within a marked code segment. Cosy also provides an interface to execute a piece of user code in the kernel. Applications like `grep`, volume rendering [10], and checksumming are the main motivation behind adding this support. These applications read large amounts of data in chunks and then perform a unique operation on every chunk. To benefit such applications, Cosy provides a secure mechanism to call a user supplied function from within the kernel.

In order to assure the secure execution of the code in the kernel, we restrict Cosy-GCC to support a subset of the C-language. Cosy-GCC ensures there are no unsupported instructions within the marked block, so complex code may need some small modifications to fit within the Cosy framework. This subset is carefully chosen to support different types of code in the marked block, thus making Cosy useful for a wide range of applications.

2.1.3 Cosy-Lib

Cosy-Lib provides a set of utility functions to insert entries into a compound. Cosy-GCC, while compiling a marked segment, inserts calls to these utility functions. So generally the functioning of Cosy-Lib is entirely transparent to the user as it is done by Cosy-GCC. But it is also possible for programmers to manually create a compound using these utility functions. It is possible to design complex or hand-optimized compounds using this facility.

2.1.4 Compound Buffer

In this section we discuss the internal representation of Cosy compound (see Figure 1). A compound is a set of entries belonging to one of the following types:

- System calls
- Arithmetic operations
- Variable assignments

- `while`, `do-while`, and `for` loops
- User provided functions
- Conditional statements
- `switch` statements
- Labels
- `goto`s (unconditional branches)

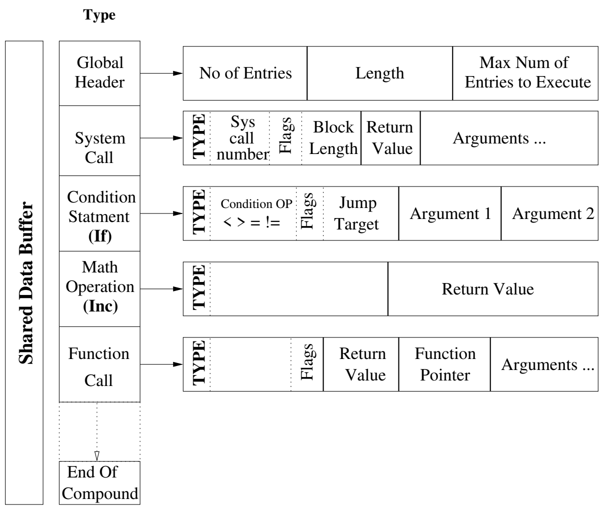


Figure 1: An Example of the Structure of a Cosy Compound

The first component of the compound buffer is the global header. It contains the number of entries in the compound, the length of the compound in words, and the maximum number of entries to execute. This entry ensures protection against unending compounds. The rest of the compound contains a set of entries with a local header followed by a number of arguments.

Each type of entry has a different structure for the local headers. Each local header has a `type` field, which uniquely identifies the entry type. Depending on the type of the entry, the rest of the arguments are analyzed. For example, if the entry is of the type "system call" then the local header contains the system call number and flags. The flags indicate whether the argument is the actual value or it is a reference to the output of another entry. The latter occurs when there are argument dependencies. If it is a reference, then the actual value is retrieved from the reference address. The local header is followed by a number of arguments necessary to execute the entry. If the execution of the entry returns any value (as in system calls, math operations, and function calls) then one position is reserved to store the result of the execution. Conditional statements affect the flow of the execution. The header of a conditional statement specifies the operator and the next instruction executed, if the condition is satisfied. Cosy-GCC resolves dependencies among the arguments and the return values, the correspondence between the label and the compound entry, and forward references for jump labels.

The overall performance of the framework depends on the efficiency of the decoder. Hence we have used several techniques like lazy caching and fast system call invocation to optimize the decoder. The first time any entry is decoded, we store the decoded value in a hash table. This makes it possible to decode an entry only once; subsequent accesses use the hashed entry. Another optimization is achieved by pushing the arguments of the system call directly onto the stack since they are packed in the Compound in the same order as they should appear on the Kernel's Stack. This makes system call invocation faster. A small piece of assembly code helps to achieve this faster invocation of system calls.

2.2 Shared Data Buffer

In this section we explain various ways that Cosy exploits zero-copy techniques by the aggregation of data-intensive code. Much work has already been done on zero-copy techniques. Cosy uses similar techniques but it tries to combine multiple techniques and provide a uniform interface to the user. Depending on the type of application, different zero-copy techniques are employed.

Cosy modifies the behavior of `copy_from_user` (copies data from the user address space into the kernel address space) and `copy_to_user` (the converse of `copy_to_user`) to enable zero-copy when the user is not interested in getting the data back into the user space. For example, when there is data dependency between a read and a following write call, Cosy uses the fast buffer to avoid the redundant copy.

Cosy also supports special versions of system calls that are commonly used. The extensive use of these system calls justifies the creation of zero-copy equivalents. We currently support zero-copy versions of `cosy_read`, `cosy_write`, and `cosy_stat`. Cosy-GCC uses these system calls automatically.

2.3 Safe Execution of a Compound

Cosy makes use of a combination of static and dynamic checks to ensure safe execution of compound. Cosy is not vulnerable to bad arguments when executing the system calls on behalf of a user process. The system call invocation by the Cosy kernel module is the same as a normal process and hence all the necessary checks are observed. However, when executing a user-supplied function, more safety precautions are needed. We describe two interesting Cosy safety features in the next sections: a preemptive kernel to avoid infinite loops, and x86 segmentation to protect kernel memory.

2.3.1 Kernel Preemption

One of the critical problems that needs to be handled while executing a user function in the kernel is to limit its execution time. To handle such situations, Cosy uses a preemptible kernel. A preemptible kernel allows scheduling of processes even when they are running in the context of the kernel. So even if a Cosy process causes an infinite loop, it is eventually scheduled out just like a normal user process. Every time a Cosy process is scheduled out, Cosy interrupts and checks the running time of the process inside the kernel. If this time has exceeded the maximum allowed kernel time, then the process is terminated. We modified the scheduler behavior to add this check for Cosy processes. The added code is minimal and is executed only for Cosy processes and hence does not affect the overall system performance. The limit on the amount of kernel time is kept sufficiently high. This high limit is not a security concern. The process even if running in kernel mode could be scheduled out and hence it does not starve other processes.

2.3.2 x86 Segmentation

To assure the secure execution of user-supplied functions in the kernel, we use the Intel x86 segmentation feature. We support two approaches.

The first approach is to put the entire user function in an isolated segment but at the same privilege level. The static and dynamic needs of such a function are satisfied using memory belonging to the same isolated segment. This approach assures maximum security, as any reference outside the isolated segment generates a protection fault. Also, if we use two non-overlapping segments for function code and function data, concerns due to self-modifying code vanish automatically. However, to invoke a function in a different segment involves overhead. Before making the function call, the Cosy kernel extension saves the current state so it is able to resume execution later. Saving the current state and restoring it back is achieved by using the standard task-switching macros, `SAVE_ALL` and `RESTORE_ALL`, with some modifications. These macros involve around 12 assembly `pushl` and `popl` instructions, each. So if the function is small and it is executed a large number of times, this approach could be costly due to the added overhead of these two macros. The important assumption here is that even if the code is executing in a different segment, it still executes at the same privilege level as the kernel. Hence, it is possible to access resources exposed to this isolated segment, without any extra overhead. Currently, we allow the isolated code to read only the shared buffer, so that the isolated code can work on this data without any explicit data copies.

The second approach uses a combination of static and dynamic methods to assure security. In this approach we restrict our checks to only those that protect against malicious memory references. This is achieved by isolating the function data from the function code by placing the function data in its own segment, while leaving the function code in the same segment as the kernel. In Linux, all the data references are resolved using the `ds` segment register, unless a different segment register is explicitly specified. In this approach, all accesses to function data are forced to use a different segment register than `ds` (`gs` or `fs`). The segment register (`gs` or `fs`) points to the isolated data segment, thus allowing access only to that segment; the remaining portion of the memory is protected from malicious access. This is enforced by having Cosy-GCC append a `%gs` (or `%fs`) prefix to all memory references within the function. This approach involves no additional runtime overhead while calling such a function, making it very efficient. However, this approach has two limitations. It provides little protection against self modifying code, and it is also vulnerable to hand-crafted user functions that are not compiled using Cosy-GCC. We are exploring compiler techniques such as self certifying code [5] to address the above concerns.

2.4 Cosy Example

In this section we illustrate some simple examples of code using Cosy. We write them once using Cosy-GCC and then without using Cosy-GCC. To improve clarity and conserve space we do not include any error checking. In these examples, we assume that the system calls without any error checking always succeed.

The following code is a simple file copy program. It reads from input file `ifile` and copies the data read to generate a duplicate file `ofile`:

```
cosy_init();

COSY_START();
ifd = open(ifile, O_RDONLY);
ofd = open(ofile, O_WRONLY);

do {
   rlen = read(ifd, buf, 4096);
   wlen = write(ofd, buf, rlen);
} while(wlen == 4096);
COSY_END();
cosy_uninit();
```

In the above program we can see that the code is almost unchanged except four new instructions. When the above code is compiled using Cosy-GCC it takes the following form:

```
1  cosy_init();
2
3  cosy_start();
4  cosy_open(&ifd, ifile, O_RDONLY);
5  cosy_open(&ofd, ofile, O_WRONLY);
6
7  cosy_do();
8     cosy_read( &rlen, ifd, buf, 4096);
9     cosy_write(&wlen, ofd, buf, 4096);
10
11 cosy_while(wlen, "==", 4096);
12 cosy_run();
13 cosy_uninit();
```

Line 1 allocates the shared buffers for the process. Line 3 clears the compound buffer. Lines 4–11 add entries into the compound. It includes a while loop around `read` and `write`. Line 12 instructs the Cosy kernel module to execute this compound. Finally, line 13 releases any buffers that are owned by this process.

3 Current Status

In this section we present benchmarks of our prototype that show the effectiveness of Cosy in improving performance of various applications. We used the following three configurations to compare the results:

1. **VAN:** This is a generic setup. This configuration does not use Cosy.
2. **COSY:** This is the Cosy configuration. It uses the Cosy framework to form and execute a compound, but does not use zero-copy techniques.

3. **COSY-FAST:** This configuration makes use of compounds and the zero-copy system calls provided by Cosy.

We performed our tests on a Intel Pentium-IV 1.7GHz machine with 64MB of RAM and a 7200 RPM 20GB ATA/100 hard drive. We repeated each test 20 times and the observed standard deviations were less that 5%. To evaluate the performance of Cosy, we report results of three benchmarks: a database simulation, `ls`, and `grep`.

Database Simulation In this benchmark we evaluate the benefits of Cosy for a database-like application. We wrote a program that seeks to random locations in a file and then reads and writes to it. The total number of reads and writes is six million. The ratio of reads to writes we chose is 2:1, matching pgmeter's [2] database workload.

We used all three configurations VAN, COSY, and COSY_FAST. We ran the benchmark for increasing file sizes. We also ran this benchmark with multiple processes to determine the scalability of Cosy in a multi-process environment. In call test, we kept the number of transactions constant at six million.

Both versions of Cosy perform better than VAN. COSY_FAST shows a 64% improvement, while COSY shows a 26% improvement in the elapsed time as seen in Figure 2. COSY_FAST is better than COSY by 38%. This additional benefit is the result of the zero-copy savings. The improvements achieved are stable even when the working data set size exceeds system memory bounds, since the I/O is interspersed with function calls.

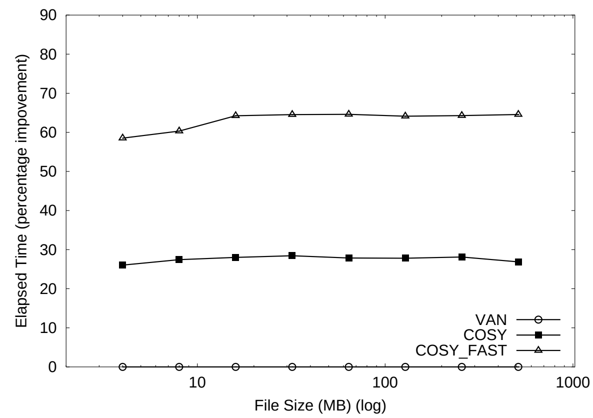
We also tested the scalability of Cosy, when multiple processes are modifying a file concurrently. We repeated the database test for 2 and 4 processes. We kept the total number of transactions performed by all processes together fixed at six million. We compared these results with the results observed for a single process. We found the results were indistinguishable and they showed the same performance benefits of 60–70%. This demonstrates that Cosy is beneficial in a multi-threaded environment as well.

ls We instrumented a Cosy `ls -l` program and compared it with the generic `ls -l` program. We use three configurations VAN, COSY and COSY-FAST for this benchmark.

For each configuration we ran this benchmark for 5000 and 50000 files and recorded the elapsed, system, and user times. We unmounted and remounted the file system between each test to ensure cold cache.

Figure 3 shows the system, user, and elapsed times taken by VAN, COSY, and COSY_FAST for listing of 5000 and 50000 files. COSY shows an 8% improvement over VAN. COSY_FAST performs 85% better than VAN for both the cases. The results indicate that Cosy performs well for small as well as large workloads, demonstrating its scalability.

grep To find out the effect of Cosy on data-intensive applications we instrumented `grep` with Cosy. This benchmark reads the files in a specified directory and executes a user provided function that searches the buffer for a given string. We recorded results for increasing sizes of data. We used all the three configurations mentioned at the start of this section.



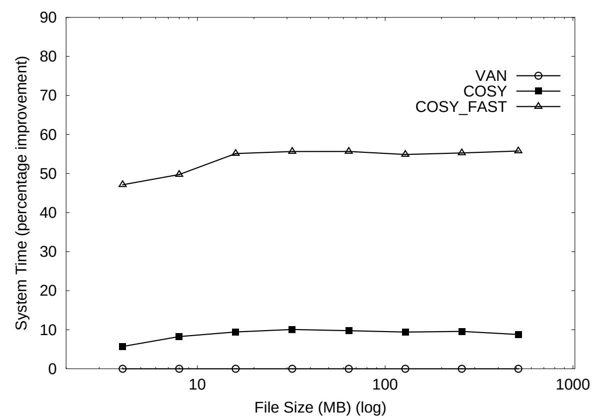


Figure 2: Elapsed and system time percentage improvements for the Cosy database benchmark (over VAN*).*

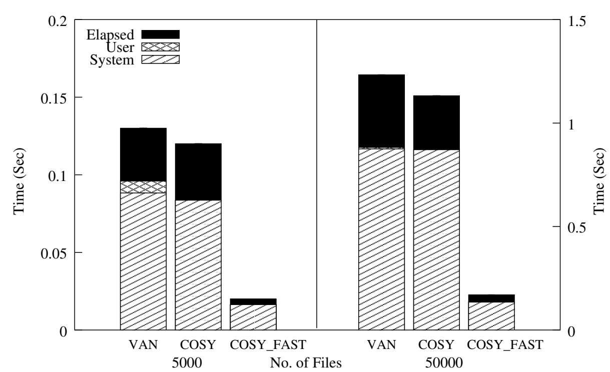


Figure 3: Elapsed, system, and user times for the Cosy `ls -l` *benchmark. Note that the left and right sides of the graph use different scales.*

The Cosy version of `grep` with zero-copy performs 20% better than the normal version (see Figure 4). The nonzero-copy version also shows improvement of 15%.

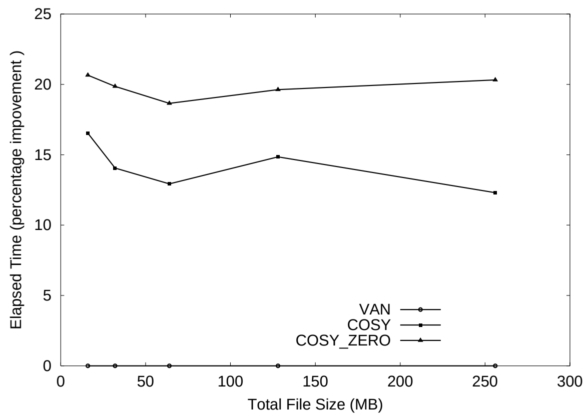


Figure 4: Elapsed time % improvement for `grep`

The 5% difference between the two flavors of Cosy justify the use of special zero-copy calls to further improve performance. This improvement substantiates our claim that moving the user function into the kernel can give us additional performance benefits.

4 Conclusions

In our work we introduce a safe yet efficient mechanism to execute bottleneck code segments of user applications, in kernel mode. We provide various zero-copy techniques that benefit different user level applications. Thus we show the applicability of Cosy under different environments. For user convenience we provide an automated mechanism to form a compound out of user-marked code. The marked code can contain loops, system calls, arithmetic operations and even some simple functions. Thus a wide range of code can be moved to the kernel transparently. Cosy supports a subset of a widely-used language, namely C, making Cosy easy to work with. We have prototyped Cosy on Linux, which is a commonly-used operating system. Many widely used user applications exist for Linux. We show performance improvements for these commonly-used applications without compromising safety.

Our benchmarks prove the usefulness and effectiveness of compound system calls. We show a speed improvement of 20–80% depending on the type of application.

4.1 Future Work

The Cosy work is an important step toward the ultimate goal of being able to execute unmodified Unix/C programs in kernel mode. The major hurdles in achieving this goal are safety concerns.

We plan to explore heuristic approaches to authenticate untrusted code. The behavior of untrusted code will be observed for some specific period and once the untrusted code is considered safe, the security checks will be dynamically turned off. This will allow us to address the current safety limitations involving self-modifying and hand-crafted user-supplied functions.

Intel's next generation processors are designed to support security technology that will have a protected space in main memory for a secure execution mode [4]. We plan to explore such hardware features to achieve secure execution of code in the kernel with minimal overhead.

To extend the performance gains achieved by Cosy, we are designing an I/O-aware version of Cosy. We are exploring various smart-disk technologies [7] and typical disk access patterns to make Cosy I/O conscious.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[2] R. Bryant, D. Raddatz, and R. Sunshine. PenguinoMeter: A New File-I/O Benchmark for Linux. In *Proceedings of the 5th Annual Linux Showcase & Conference*, pages 5–10, Oakland, CA, November 2001.

[3] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. Technical Report CMU-CS-00-117, Carnegie Mellon University, March 2000.

[4] H. B. Pedersen. Pentium 4 successor expected in 2004. *Pcworld*, October 2002. www.pcworld.com/news/article/0,aid,105882,00.asp.

[5] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the architecture of the VINO kernel. Technical Report TR-34-94, EECS Department, Harvard University, 1994.

[6] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3010, Network Working Group, December 2000.

[7] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of First USENIX conference on File and Storage Technologies*, March 2003.

[8] T. Maeda. Safe Execution of User programs in kernel using Typed Assmebly language. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml, 2002.

[9] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 203–216, Asheville, NC, December 1993. ACM SIGOPS.

[10] C. Yang and T. Chiueh. I/O conscious Volume Rendering. In *IEEE TCVG Symposium on Visualization*, May 2001.

[11] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.