Improving Application Performance Through System Call Composition

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

Technical Report FSL-02-01

Abstract

Long-running server applications can easily execute millions of common data-intensive system calls each day, incurring large data copy overheads. We introduce a new framework, Compound System Calls (Cosy), to enhance the performance of such applications. Cosy provides a mechanism to safely execute data-intensive code segments in the kernel. Cosy encodes a C code segment containing system calls in a compound structure. The kernel executes this aggregate compound directly, thus avoiding data copies between user-space and kernel-space. With the help of a Cosy-GCC compiler, regular C code can use Cosy facilities with minimal changes. Cosy-GCC automatically identifies and encodes zero-copy opportunities across system calls. To ensure safety in the kernel, we use a combination of static and dynamic checks, and we also exploit kernel preemption and hardware features such as x86 segmentation. We implemented the system on Linux and instrumented a few data-intensive applications such as those with database access patterns. Our benchmarks show performance improvements of 20–80% for non-I/O bound applications.

1 Introduction

Applications like FTP, HTTP, and Mail servers move a lot of data across the user-kernel boundary. It is well understood that this cross-boundary data movement puts a significant overhead on the application, hampering its performance. For data-intensive applications, data copies to user-level processes could slow overall performance by two orders of magnitude [33]. For example, to serve an average Web page that includes five external links (for instance images), a Web server executes 12 read and write system calls. Thus a busy Web server serving 1000 hits per second will have executed more than one billion costly data-intensive system calls each day.

One method to address the problem of data movement is to extend the functionality of existing OSs to satisfy the needs of the application [2, 9, 22, 28]. The problem with such methods is that they require an entirely new framework with a special OS or special languages to assure the safety of the system. Another way to reduce data copies is simply to minimize the number of times context switches happen. For example, reading files in large chunks instead of small ones can reduce the number of times the `read` system call is invoked to read a file entirely. A third method, often used in networking, is to aggregate network or protocol messages together and send them as one larger message. NFSv4 [24] defines a compound message as a packed sequence of RPC messages, each specifying a single NFSv4 operation. In this way, an NFSv4 client could send a distant server one message that encodes several requests, thus reducing network latency and setup time.

Cosy provides a framework that allows user applications to execute their data-intensive code segments in the kernel, thereby avoiding data copies. Cosy aggregates the system calls and intermediate code belonging to a data-intensive code segment to form a compound. This compound is passed to the kernel via a new system call (`cosy_run`), which decodes the compound and executes the encoded operations, avoiding data copies.

Zero-copy techniques, sometimes known as fast-path architectures, reduce the number of times data is copied [1, 12, 13, 16]. Cosy employs zero-copy techniques in three places. First, the Cosy buffer used to encode a compound is itself a physical kernel memory segment mapped to the user space application; this way both the kernel and the user process can read from or write to this shared memory space without an explicit copy. Second, the Cosy system allows a user application to allocate additional contiguous physical kernel memory that is also mapped to the user application. This memory, called a *Cosy shared data buffer*, can be used like any other `malloc`ed memory, only that using this memory does not require copying between user processes and the kernel; this is particularly useful for system calls that run frequently (e.g., `stat`) or copy a lot of data between the user and kernel address spaces (e.g., `read` or `write`). Third, Cosy allows compounded system calls to directly share system call arguments and return values. For example, the file descriptor number returned by `open` can be passed to a `read` call so it can operate on that opened file; and a memory buffer (whether a Cosy shared buffer or not) used by a `read` call could be passed to a subsequent `write` call, which can then use it directly.

We provide Cosy-GCC, a modified version of GCC 3.2, to automatically convert data-intensive code into a compound. The user just needs to mark the data-intensive code segment and Cosy-GCC converts the code into a compound at compile time. Cosy-GCC also resolves dependencies among parameters of Cosy statements and encodes this information in the compound. Cosy uses this information to reduce data copies while executing the compound in the kernel.

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 [28]. Cosy uses a combination of static and runtime approaches to assure safety in the kernel. Cosy explores various hardware features along with software techniques to achieve maximum safety without adding much overhead.

We have prototyped the Cosy system on Linux. We conducted a series of general-purpose benchmarks and micro-benchmarks comparing regular user applications to those that use Cosy. We found overall performance improvements of Cosy to be up to 20–80% for common non-I/O bound user applications.

The rest of this paper is organized as follows. Section 2 describes the design of our system and includes safety features in detail. We discuss interesting implementation aspects in Section 3. Section 4 describes the evaluation of our system. We review related works in Section 5 and conclude in Section 6.

2 Design

Often only a critical portion of the application code suffers due to data movement across the user-kernel boundary. Cosy encodes the statements belonging to a bottleneck code segment along with their parameters to form a *compound*. When executed, this compound is passed to the kernel, which extracts the encoded statements and their arguments and executes them one by one, avoiding extraneous data copies. We designed Cosy to achieve maximum performance with minimal user intervention and without compromising security. The three primary design objectives of Cosy are as follows:

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

Safety We use various security features involving kernel preemption and hardware-specific features such as Intel's segmentation, to assure a robust safety mechanism even in the face of errant or malicious user programs. We use a combination of static and dynamic checks to assure safety in the kernel without adding much runtime overhead. We discuss safety design issues in Section 2.6.

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 easy.

2.1 Architecture

To facilitate the formation and execution of a compound, Cosy provides three components: Cosy-GCC, Cosy-Lib and the Cosy Kernel Extension. Users need to identify the bottleneck code segments and mark them with the Cosy specific constructs `COSY_START` and `COSY_END`. This marked code is parsed and the statements within the delimiters are encoded into the Cosy language. We call this intermediate representation of the marked code segment a *compound*. Encoded statements belonging to a compound are called Cosy *operations*.

The Cosy system uses two buffers for exchanging information. First, a compound is encoded in a *compound buffer*. Second, Cosy uses a *shared buffer* to facilitate zero-copying of data within system calls and between user applications and the kernel.

2.2 Cosy-GCC

Cosy-GCC automates the tedious task of extracting Cosy operations out of a marked C-code segment and packing them into a compound, so the translation of marked C-code to an intermediate representation is entirely transparent to the user.

Cosy-GCC also resolves dependencies among parameters of the Cosy operations. Cosy-GCC determines if the input parameter of the operations is the output of any of the previous operations. It is necessary to encode this dependency, as the real values of the parameters are not known until the operations are actually executed. While parsing the marked code, Cosy-GCC maintains a symbol table of output parameters of the operations and labels. It compares each of the input parameters of any new operation against the entries in the symbol table to check for any dependencies. This dependence is marked in the `flags` field (Section 2.3.1) of the compound buffer. For conditional statements or jumps, the control flow within the compound may vary depending on the outcome of the conditional statement. Cosy-GCC determines the next operation to execute in case that a branch is taken or not taken. To resolve forward references in such cases, Cosy-GCC uses a symbol table.

Cosy supports loops (i.e., `for`, `do-while`, and `while`), conditional statements (i.e., `if`, `switch`, `goto`), simple arithmetic operations (i.e., increment, decrement, assignment, add, 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 [31], 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 completely 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.

2.3 Cosy-Lib

The Cosy library provides utility functions to create a compound. Statements in the user-marked code segment are changed by the Cosy-GCC to call these utility functions. So the functioning of Cosy-Lib and the internal structure of the compound buffer are entirely transparent to the user.

Cosy-Lib is also responsible for maintaining the shared data buffer. The library extends the malloc library to efficiently handle the shared data buffer. User applications that wish to exploit zero-copy can manage memory from the shared data buffer with the `cosy_malloc` and `cosy_free` functions provided by our library.

2.3.1 Structure of a Compound

In this section we describe the internal structure of a compound, which is stored within the compound buffer (see Figure 1). A compound is the intermediate representation of the marked code segment and contains a set of operations belonging to one of the following types:

- System calls
- Arithmetic operations
- Variable assignments
- `while` and `do-while` loops
- Optimized `for` loops (See Section 3)
- User provided functions
- Conditional statements
- `switch` statements
- Labels
- `goto`s (unconditional branches)

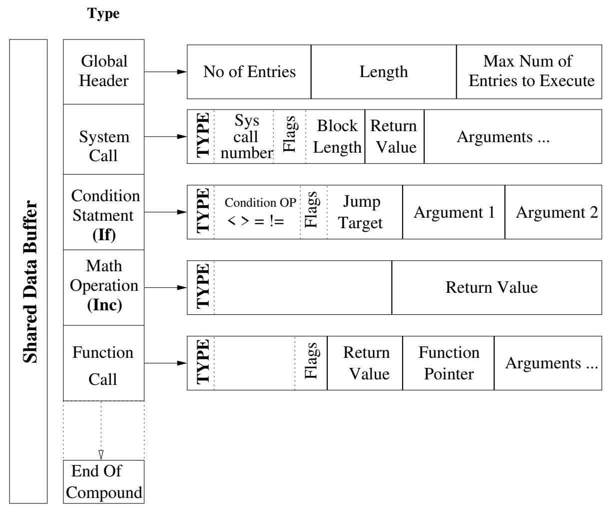


Figure 1: Internal structure of cosy compound: An example of a compound of a system call, a conditional, an assignment, and an addition.

The compound buffer is shared between the user and kernel space. The operations that are added by the user into the compound are directly available to the Cosy Kernel Extension without any data copies. The first field of a compound is the global header that contains the total number of operations encoded in the compound. The "End of Compound" field is required since each Cosy operation may occupy a variable length. The compound also contains a field to set the upper limit on the maximum number of operations to be executed. This limit is necessary to avoid infinite loops inside the kernel. The remaining portion of the compound contains a number of operations. The structure of each operation is of the following form: a local header followed by a number of arguments needed for the execution.

Each type of operation has a different structure for the local headers. Each local header has a `type` field, which uniquely identifies the operation type. Depending on the type of the operation, the rest of the arguments are analyzed. For example, if the operation is of the type "system call," then the local header will contain the system call number and flags. The flags indicate whether the argument is the actual value or a reference to the output of some other operation. 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 operation. If the execution of the operation returns any value (e.g., a system call, math operation, or a function call) one position is reserved to store the result of the operation. Conditional statements affect the flow of the execution. The header of a conditional statement specifies the operator, and if the condition is satisfied, the next instruction is then executed. Cosy-GCC resolves dependencies among the arguments and the return values, the correspondence between the label and the compound operation, and forward references for jump labels.

2.4 The Cosy Kernel Extension

The Cosy kernel extension extracts the operations encoded in a compound and executes them one by one. The Cosy kernel extension provides three system calls:

- `cosy_init` allocates both the compound buffer and the shared data buffer that will be used by the user and the kernel to exchange encoded compounds and return results.
- `cosy_run` decodes the compound and executes the decoded operations one by one.
- `cosy_free` releases any resources allocated for Cosy on behalf of this process. This is optional: the kernel will also garbage collect such resources upon termination of the process.

2.4.1 Executing the Compound

The Cosy kernel extension is the heart of the Cosy framework. It decodes each operation within a compound buffer and then executes each operation in turn.

The normal behavior of any user application is to make system calls, and based on the results, decide the next set of system calls to execute; so the sequence in which the system calls are executed is not constant. Therefore, it is not sufficient to execute the system calls belonging to a compound in the order that they were packed. After executing one system call, the Cosy kernel extension checks the result and decides the next system call to execute. The Cosy framework supports programming language constructs such as loops, conditionals, and math operations. This way the user program can encode conditional statements and iterative instructions into the compound. The Cosy kernel extension executes the system calls in the sequence that they were packed unless it reaches a condition or loop statement. At that point, it determines the next call to execute depending on the result of the conditional statement. Cosy also supports another mode, where the Cosy kernel extension exits on the first failure of any Cosy operation. This mode is useful while executing a long loop in the kernel that has error checking after the loop.

We limit the framework to support only a subset of C. One of the main reasons is safety, which we discuss in Section 2.6. Another issue is that extending the language further to support more features may not increase performance because the overhead to decode a compound increases with the complexity of the language. The savings may not be manifested as a result of this overhead.

Efficient decoding of the compound is required to achieve our performance goals. To achieve this, we exploit various optimizations. To make decoding efficient, Cosy uses lazy caching of decoded data. Thus the first time any operation is visited it is decoded and this decoded operation is stored in a hash table. The next time the same operation has to be evaluated, decoding is avoided. During system call execution, the arguments of the system calls are pushed directly onto the stack and then the system call function is called using a small amount of assembly code. This avoids any intermediate copying of arguments from the compound buffer to local buffers and hence speeds up the invocation of system calls. The savings in the decoding time and the time to invoke a system call help to minimize Cosy's overhead.

2.5 Zero-Copy Techniques

Many common system calls can realize performance improvements through zero-copy techniques. `read` and `write` are especially good candidates as they normally copy large amounts of data between the kernel and user space. Calls like `stat` can also benefit from zero-copy techniques because they are invoked very often.

The area where user programs can achieve the largest savings from the shared data buffer is system time. So this method is particularly useful when the data copies are not I/O bound. The Cosy framework uses the shared data buffer to support zero-copy data transfers by modifying the behavior of `copy_to_user` and `copy_from_user`. Many system calls including `read` and `write` utilize these copy calls and will enjoy the benefits of using the shared data buffer by avoiding redundant data copies.

Whenever a user makes a `read` system call using the shared buffer, Cosy checks for the use of a shared buffer and skips the `copy_to_user` call avoiding a data copy back to user-space. Cosy stores the physical address of the page that contains the read data in `struct task`. After that, if the user makes any call that uses the same shared buffer, then the stored physical address is provided to that call. For example, if the user makes a `write` call which also uses the same shared buffer, Cosy uses the stored physical address in `copy_from_user`.

As the check for zero-copy is performed in `copy_to_user` and `copy_from_user`, which are generic calls, any system call that performs data copies receives the benefit of zero-copy by using the shared buffer. Cosy is not skipping any validation checks, so safety is not violated. So in the worst case a bad address provided by a buggy or malicious program will be passed to the system call. The system call still checks the validity of all arguments, so a segmentation fault will be generated as normal.

System calls like `stat` copy information about a file from kernel kernel to user space. In this case Cosy saves copies by allocating the `stat` buffer in physical kernel memory that is shared between the kernel and the user process. Cosy can take this one step further and exploit the relationships between system calls that share data buffers. For instance, data is often read from one file and written to another. If we allocate this buffer in physical kernel memory, then we can share it directly between calls within the kernel.

Cosy supports special versions of existing system calls to enable zero-copy by default using the shared data buffer. These system calls are accessible only through Cosy. Currently we support `stat`, `read`, and `write` as described in Section 3. When applications use memory allocated using `cosy_malloc` and use it in `read`, `write`, or `stat` system calls, Cosy-GCC detects the possible optimization and converts these calls to their zero-copy versions.

2.6 Safety Features

Cosy applies runtime bound checking to prevent possible overruns of the shared buffer. Cosy is not vulnerable to bad arguments when executing 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 performed. However, when executing a user-supplied function, more safety precautions are needed. Cosy makes use of the hardware and software checks provided by the underlying architecture and the operating system to do this efficiently. 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.

2.6.1 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. 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.

2.6.2 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 to resume execution. 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 `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.

2.7 Cosy Examples

To understand the phases of Cosy in greater detail we demonstrate a simple C program that reads a file using Cosy features, and then show its internal representation after the Cosy-GCC modification.

The C code below reads an input file `name` until the end of the file. For simplicity we do not include any error checking in this example:

```
COSY_START;
fd = open(name,f,m);
do {
   rln = read(fd,bf,ln);
} while(rln == ln);
close(fd);
COSY_END;
```

The code segment is marked with `COSY_START` and `COSY_END`. When this program is compiled using Cosy-GCC it replaces the statements with calls to Cosy-Lib functions to add the statements into a compound. We show the converted code after the Cosy-GCC compilation below:

```
1 cosy_add(&fd,NR_open,0,name,0,f,0,m);
2 cosy_do();
3   cosy_add(&rln,NR_read,1,fd,0,bf,0,ln);
4 cosy_while(1,rln,"==",0,ln);
5 cosy_add(__NR_close,1,fd);
6 cosy_run();
```

Statements 1 through 5 add operations to the compound. Statement 6 is a Cosy Kernel Extension call that informs the Cosy kernel to execute the compound. In statement 3, the third parameter is a flag indicating that the parameter value `fd` is not known and should be retrieved from the output of the first operation (`open`).

3 Implementation

We implemented a Cosy prototype on Linux 2.4.20. This section highlights the following five implementation issues: kernel changes, shared buffers, zero-copy, faster system call invocation, and loop constructs.

Kernel Changes and Maintenance The number of lines of code that we have changed is a good indicator of the complexity of our system. The kernel proper only needs a 37 line patch, which is only necessary so that the Cosy kernel module can interface with the static kernel to modify the task structure. Our system consists of three components: Cosy-GCC, a user-level library, and a kernel module. The patch to GCC is 600 lines, the kernel code is 1877 lines, and the library is 4002 lines. Most of the kernel code handles the decoding of the compound call. To make it easier for the user to write programs using Cosy we provide an interface that for a subset of C. To support this feature we created a small database containing the list of all the system calls. We auto-generate the code to support these system calls. This auto-generation makes code development and maintenance simpler.

The changes to the task structure involve addition of three fields. Since Cosy allocates kernel buffers for each process. Since kernel buffers are a scarce resource, it is the responsibility of the allocator to release this memory after the process's termination. To facilitate this resource reclamation we added a field into the task structure that points to a structure containing all the kernel allocated addresses, and a field that contains a pointer to a cleanup function that will release these resources. The third field contains a timer that indicates the total amount of time that this Cosy compound has been executing in the kernel. This timer is used to terminate the process when the process exceeds its allowed time span.

Shared Buffers There are some standard mechanisms that enable sharing of data between user and kernel space. We explored two such approaches in great detail to determine which one gives us the fastest way of sharing data. The first approach was by using `kiobufs`. `Kiobufs` facilitate user-mode DMA. A user application allocates a buffer in user space and passes the virtual address to the kernel. The kernel determines the physical address of the page and stores it in the `kiobufs`. Whenever the kernel wishes to access this data it can just look into the `kiobuf` for the physical address of the page. The limitation of this approach is that multiple pages may not be allocated contiguously in the physical address space. To compensate for this, the kernel needs to check which page is under consideration and determine its physical address. This would decrease performance for large memory segments. Instead, we chose to have the kernel `kmalloc` a set of pages in the memory and map these pages to the process' address space. As `kmalloc` always returns physically contiguous memory, both the user and the kernel can access our shared buffers sequentially.

Zero-Copy Supporting zero-copy without major kernel changes was the biggest challenge we faced. We explored different options to do a zero-copy `read` and `write` operation. It is possible to modify the `read` and `write` system calls to make them support zero-copy. We can also implement different versions of these calls which support the zero-copy data transfer. Both of the above mentioned solutions are specific to particular calls. We adopted a more generic approach to implement zero-copy transfer. Both `read` and `write` and their variants that deal with data copy make use of the macros `copy_to_user` and `copy_from_user`. To provide a generic solution, we modified these two macros. In `copy_to_user` we avoid the data copy and instead save the source page address belonging to the disk block. When the read data is to be written, `sys_write` calls `copy_from_user` to copy the data from the user buffer. At this point the address `copy_to_user` stored in the task structure is fed to the write as the source for data.

`stat` requires different techniques than read and write because it returns only a 64 byte structure and the mapping techniques we use for `copy_to_user` are less efficient with for segments of memory. Cosy defines `cosy_stat`, which is a special version of `sys_stat` that operates only on shared buffers. `sys_stat` reads some information about a requested file into a kernel buffer and then copies it to a supplied user buffer. Unlike `sys_stat`, Cosy_stat writes directly into the user-supplied fast buffer, which is accessible to the user application. Thus Cosy_stat avoids one data copy of the stat buffer. The user can allocate a chunk in the fast buffer by calling `cosy_malloc`, which is then used by `cosy_stat` to execute a zero-copy stat.

Faster System Call Invocation We use assembly code to speedup the invocation of system calls. The system call arguments are aligned in the shared buffer, in exactly the same order as they have to be pushed onto the kernel stack. We push them directly onto the stack using the IA32 assembly instruction `pushl` *param* and then call the function belonging to the system call using another assembly instruction `call` *sys_call_ptr*. This way the argument handling is more efficient. Copying of arguments into any of the local buffers is avoided. There are only 25 lines of assembly code consisting of only simple arithmetic, push, and jump instructions.

Loop Constructs Cosy has three forms of loops: `while`, `do-while`, and `for`. In the most general case a `for` loop is converted to an equivalent `while` loop, but many `for` loops are of the form:

```
for (i = I; i conditional C; i += N)
```

In this case converting the `for` loop to a `while` loop will require three Cosy operations: the initialization (i = I), the loop (`while` (i conditional C)), and the addition (i += N). However, if the loop is in this common form, we use a special `for` operator, which encodes the parameters I, C, and N into a single operation. Using a single Cosy operation avoids decoding the addition operation during each loop iteration.

4 Evaluation

To evaluate the behavior and performance of compound system calls we conducted extensive benchmarking on Linux comparing the standard system call interface to various configurations using Cosy. In this section we (1) discuss the benchmarks we performed using these configurations, (2) demonstrate the overhead added by the Cosy framework using micro-benchmarks, and (3) show the overall performance on general-purpose workloads.

4.1 Experimental Setup

We ran each benchmark using a subset of the following three configurations:

1. **VAN:** A vanilla setup where benchmarks use standard system calls, without Cosy.
2. **COSY:** A modified setup where benchmarks use the Cosy interface to form compounds and send them to the kernel to be executed.
3. **COSY-FAST:** A setup identical to the COSY setup except that it uses a fast shared buffer to avoid memory copies between user-space and kernel-space.

Our experimental testbed was a 1.7GHz Intel Pentium 4 machine with 128MB of RAM and a 36GB 10,000 RPM IBM Ultrastar 73LZX SCSI hard drive. All tests were performed on an Ext2 file system, with a single native disk partition that was the size of our largest data set to avoid interactions with rotational delay [8].

We installed the vanilla Linux 2.4.20 kernel and applied the Cosy kernel patch and the kernel preemption patch. All user activities, periodic jobs, and unnecessary services were disabled during benchmarking. We measured Cosy performance for a variety of CPU speeds. However, we only report the results for the 1.7GHz Pentium 4 because the results are not significantly different for the other CPU speeds.

We ran each experiment at least 20 times and measured the average elapsed, system, user, and I/O (wait) times. Finally, we measured the standard deviations in our experiments and found them to be small: less than 5% of the mean for most benchmarks described. We report deviations that exceeded 5% with their relevant benchmarks.

4.2 Cosy Overhead

Using the configurations mentioned in Section 4.1 we performed a `getpid` micro-benchmark to evaluate the efficiency and overhead of the Cosy framework. This benchmark shows the overhead involved with forming a compound and executing it using the Cosy framework. We chose `getpid` because it performs a minimal amount of work in the kernel.

We ran this benchmark for the VAN and COSY setups. We omitted the COSY_FAST configuration because the fast buffer does not serve a purpose for `getpid`.

The VAN benchmark program executes a number of independent `getpid` system calls within a `for` loop. The COSY setup constructs a `for` loop to be decoded by `cosy_run` and evaluated in the kernel. For each test we ran the benchmark for an exponentially increasing number of getpid calls: 2, 4, 8, ..., 256. This helped us measure the scalability of the Cosy framework.

Figure 2 shows that COSY is more efficient than VAN. The improvements range from 12–90% in elapsed time, 36–85% in system time, and -10–100% in the user time.

When the number of getpids is 2 or less COSY shows 10% penalty in user time. This is because COSY has an overhead of creating the compound in user space. COSY adds two operations in the compound: a for loop and a system call getpid. Even if the number of getpids is increased, the size of the compound remains the same and hence the user-space overhead remains the same. On the other hand, as the number of getpids increases, the amount of work VAN does in user space increases linearly. Hence, initially when the number of getpids is small (less than 3), VAN looks better in user time; but, as the number of getpids increases, COSY performs better than VAN.

COSY shows improvement in system time, even though it is decoding and executing a loop inside the kernel. This is because the loop overhead is less costly than context switching. The benchmark indicates that even after paying the overhead of decoding a loop, COSY performs 36–85% better than VAN. The results indicate that the decoding overhead in Cosy is minimal.

Elapsed time results always show improvement, irrespective of the number of getpids. Thus, even if COSY has some overhead for small compounds in user time, the system time savings more than compensate, resulting in overall performance improvement.

4.3 General Purpose Benchmarks

Using the configurations defined in Section 4.1 we conducted four general purpose benchmarks to measure the overall performance of the Cosy framework for general-purpose workloads: database, Bonnie, ls, and grep.

Database Simulation In this benchmark we find 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. We followed similar techniques as used by Bonnie [7] and pgmeter [4] to simulate the database access patterns. The ratio of reads to writes we chose is 2:1, matching pgmeter's database workload.

We used the three configurations VAN, COSY, and COSY_FAST. The Cosy versions of the benchmark program create a compound and executes it for a user-specified number of iterations. This compound executes a function to generate a random number, for use as an offset into the file. The next operation in the compound is to seek to this random offset, and then read from that location. On every alternate iteration, the compound executes a write after the read. COSY_FAST exploits zero-copy while reading and writing the same data, while COSY is the non-zero-copy version of the same benchmark.

We ran the benchmark for increasing file sizes. We kept the number of transactions constant at six million. We also ran this benchmark with multiple processes to determine the scalability of Cosy in a multiprocess environment.

Bonnie We used the Bonnie benchmark [7] to measure the benefits of Cosy's zero-copy techniques. Bonnie is a file system test that intensely exercises both sequential and random reads and writes. Bonnie has three phases. First, it creates a file of a given size by writing it one character at a time, then it rewrites the file in chunks of 4096, and then it writes the same file one block at a time. Second, Bonnie reads the file one character at a time, then a block at a time; this can be used to exercise the file system cache, since cached pages have to be invalidated as they get overwritten. Third, Bonnie forks 3 processes that each perform 4000 random lseeks in the file, and read one block; in 10% of those seeks, Bonnie also writes the block with random data. This last phase of Bonnie simulates a random read+write behavior, often observed in database applications.

We modified Bonnie to use Cosy. In the first phase we modified the block write and rewrite sections. We skip the first section where Bonnie writes to a file using putc since it is a glibc function that uses buffered I/O and hence not applicable to Cosy. In the second phase we modified the block read section. We did not modify the third phase as we have demonstrated a database simulation application in the previous benchmark, and ours is more intense than Bonnie's. Our database benchmark simulates the database read+write patterns more accurately, because the number of write operations performed by Bonnie are less than that generally observed in database workloads [4]. Our database benchmark also runs for 30 seconds doing six million read+write transactions. The third phase of Bonnie executes just 4000 transactions, which takes less than one second.

For the first Bonnie phase we used two configurations, VAN and COSY_FAST, where we compare fastread and rewrite performance. COSY_FAST is useful in the first phase as both fastread and rewrite exploit the zero-copy techniques. When performing block data writes in the second phase it is not possible to save any data copies. This is why we use VAN and COSY for the second phase. We ran the benchmarks for exponentially increasing file sizes from 4–512MB.

ls Listing directory contents can be enhanced by the Cosy framework. Here we benchmarked our own Cosy ls program and compared it to a standard ls program. We ran this program with the -l option in order to force ls to make a stat system call for each file listed.

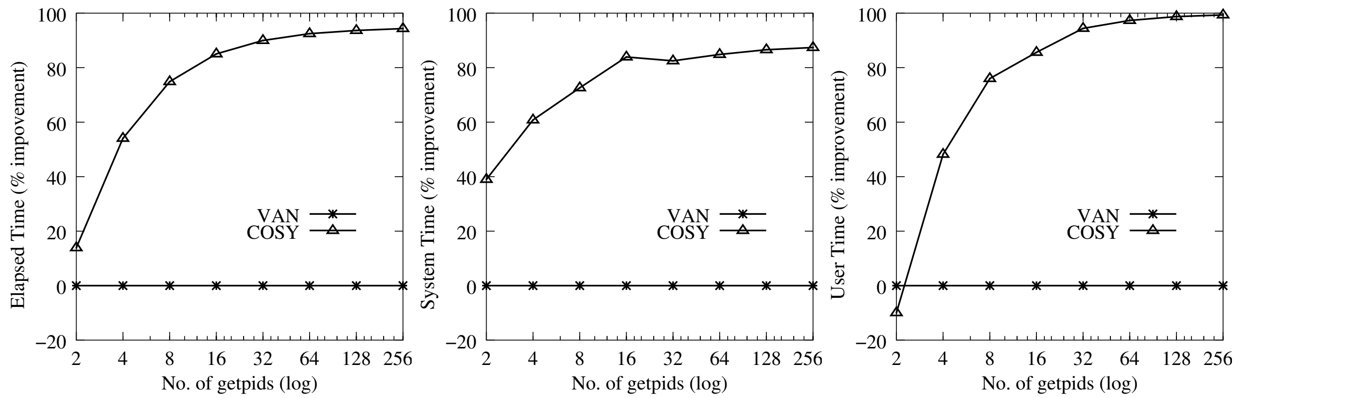


Figure 2: Elapsed, system, and user time percentage improvements of COSY *over* VAN *for getpid.*

We used all three configurations defined in Section 4.1: VAN, COSY, and COSY_FAST.

The Cosy versions of the `ls` benchmark program creates a compound that performs `getdents` and uses its results to determine the entries to be `stat`ed. This compound is then sent to the kernel for execution. The COSY_FAST benchmark uses a special `cosy_stat` system call, which is a zero-copy version of the generic `stat` system call (automatically selected by Cosy-GCC). We performed this benchmark to show the effectiveness of new Cosy system calls. We benchmarked `ls` with cold cache to test the performance of the special Cosy systems calls under a worst-case scenario.

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.

grep `grep` is another common user application that can benefit from Cosy. `grep` represents the class of applications that read a lot of data and work on that data without modifying it. In this regard, it is similar to checksumming or volume rendering applications [31].

We used three configurations VAN, COSY, and COSY_FAST to analyze the performance of `grep`. The Cosy versions of the `grep` benchmark create a compound that opens a specified file, reads each 4096 byte chunk, executes a user-supplied function that searches the chunk for a particular string, and repeats this process until an end-of-file is reached. This process is repeated for a specified number of files using a `for` loop. The difference between the two versions of these Cosy benchmarks is that COSY copies the chunk back to the user-space, while COSY_FAST works on the kernel buffer avoiding the copy back to user space.

We ran this benchmark for an increasing number of 8K files; however, we plot the graphs against the total size of data read. The total size of data varies from 128K to 2MB. We chose a file size of 8K as it is observed that most accessed files are small [20].

4.4 General Purpose Benchmarks Results

Database Simulation 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 3. 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.

Figure 4 shows the absolute elapsed and system times for the database benchmark. We show absolute times to understand the extent of saving achieved by the application. COSY_FAST is 20 seconds faster than VAN and 12 seconds faster than COSY. In the user time both versions of Cosy perform better than VAN, saving over 6 seconds. We do not report the I/O (wait) time for this test, because the I/O is interspersed with CPU usage, and hence insignificant (less than 1%).

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 multiprocessor environment as well.

Bonnie We explain the results of Bonnie in three phases: fastread, rewrite, and fastwrite.

As shown in Figure 5(a), the Cosy version of `fastread` showed a considerable performance improvement of 80%, until it is bound by the amount of

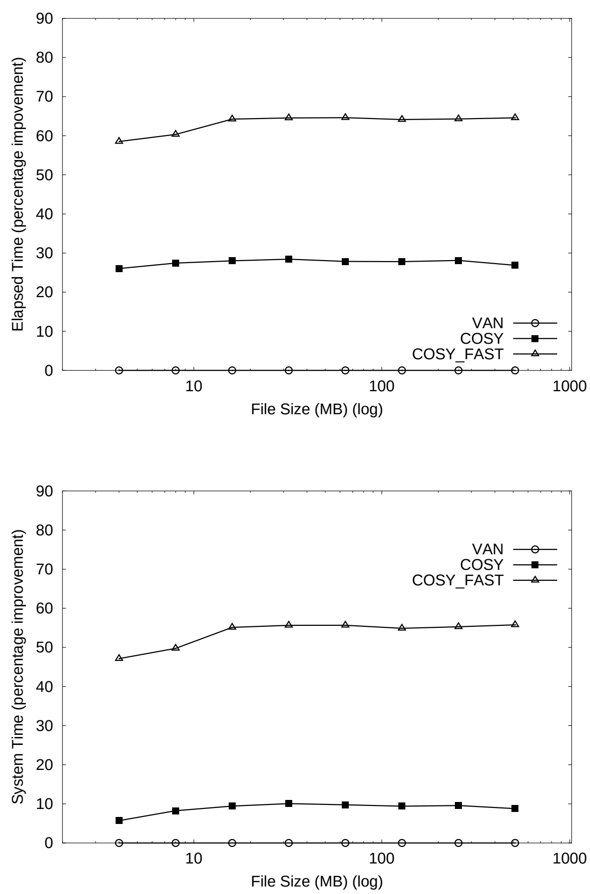


Figure 3: Elapsed and system time percentage improvements for the Cosy database benchmark (over VAN*).*

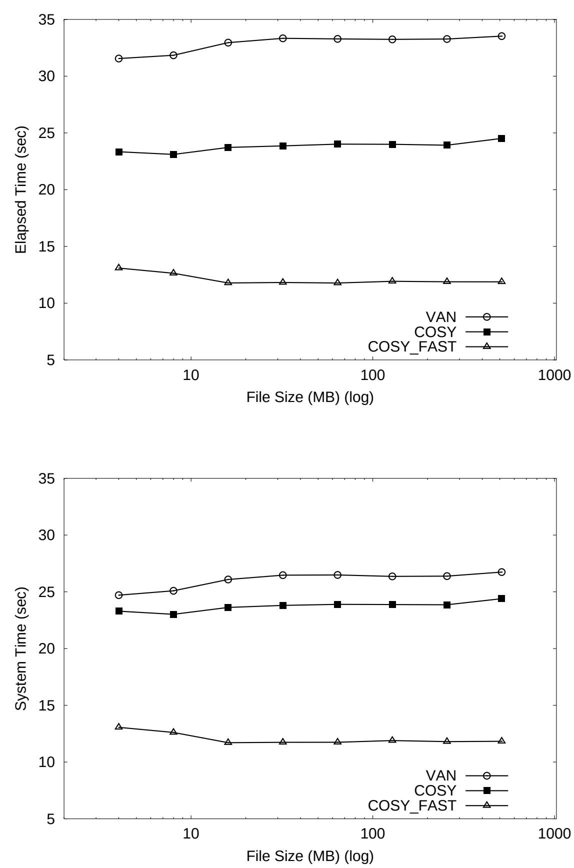


Figure 4: Absolute elapsed and system times for the Cosy database benchmark.

available memory (in this case 128MB). Cosy provides savings in system time by avoiding unnecessary data copies. When an application triggers heavy I/O activity, the savings achieved by Cosy become less significant when compared to I/O time. Hence, we observed the drop in the performance improvement at 128MB.

The `rewrite` phase using Cosy shows performance benefits of 30–90% over the VAN. COSY_FAST exploits the zero-copy technique to bypass the data copy back to the user. The major improvement comes from savings in system time. The drop in improvement occurs when the benchmark begins to fill up the memory as indicated by Figure 5(b).

Figure 5(c) indicates that the Cosy version of `fastwrite` is better than VAN by 45–90% for file sizes up to 64MB. When the benchmark begins to fill available system memory (128MB), the performance gains observed in Cosy are overshadowed by the increasing I/O time and by Linux's page flushing algorithm (suspend all process activity and purge caches aggressively). `fastwrite` is an I/O-intensive benchmark. Currently, Cosy is not designed to help with I/O; hence, as the I/O activity increases, the Cosy performance benefits become less significant.

ls Figure 6 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.

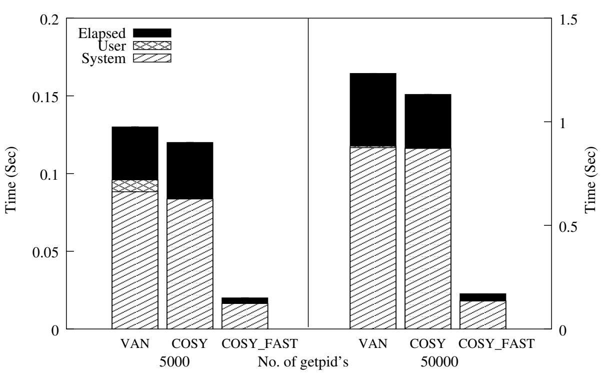


Figure 6: Elapsed, system, and user times for the Cosy `ls -l` benchmark. Note the left and right side of the graph use different scales.

System time savings for COSY are small when compared to COSY_FAST. COSY_FAST uses the zero-copy version of `stat` and hence it is faster than the non-zero-copy version (COSY). We performed this benchmark with a cold cache. The improvements in the COSY_FAST results indicate that Cosy is useful even when the data is not present in memory, provided the amount of I/O involved is small.

grep Cosy versions of `grep` perform better than VAN. Figure 7 shows that COSY is 13% better than VAN and COSY_FAST is 20% better than VAN.

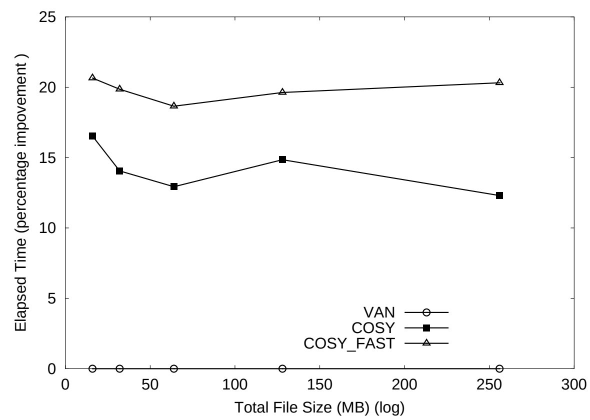


Figure 7: Elapsed time percentage improvements for the Cosy `grep` benchmark (compared to VAN).

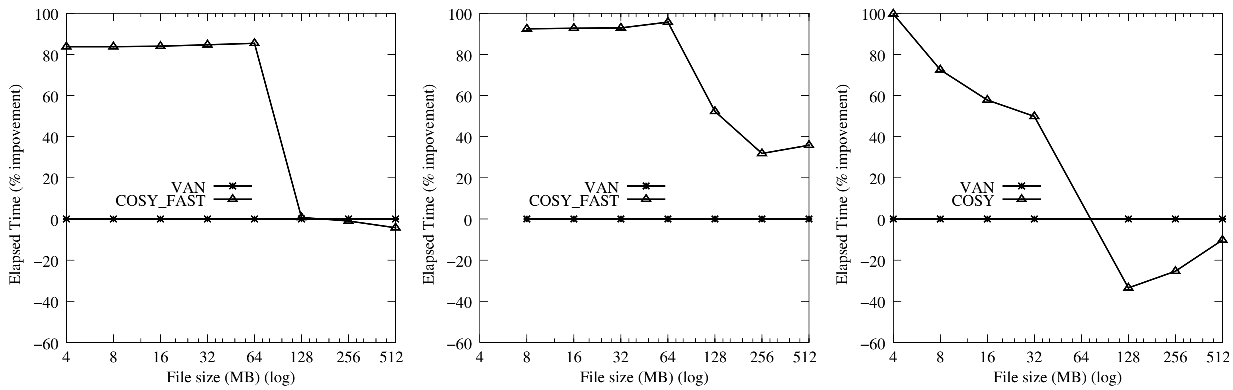
The system time for the Cosy versions of `grep` is large compared to VAN. System time is primarily composed of three components, (1) the time taken by in-memory data copies, (2) the time taken by the user-supplied function, and (3) other system call and Cosy overhead. In the Cosy versions of `grep`, major chunks of code are executed in the kernel, resulting in an increase in the system time taken by user functions. However, the user time for the Cosy versions of `grep` is reduced by that same amount. The savings in data copies (a component of system time) and in user time more than compensate for the increase in system time due to the user function. From this result we can conclude that even if the system time increases, the overall performance can be improved as a result of savings in data copies and user time.

5 Related Work

The related work section is divided into three parts: composing multiple operations into a single call, zero-copy techniques, and security techniques for executing user code in kernel mode.

5.1 Composition of Operations

The networking community has long known that better throughput can be achieved by exchanging more data at once than repeatedly in smaller units. Analysis of NFSv2 traffic has shown that a large fraction of RPC calls use a READDIR operation followed by many GETATTR operations [26, 32]. To improve its performance, the NFSv3 protocol includes a new RPC pro-



(a) Fastread (b) Rewrite (c) Fastwrite

Figure 5: Elapsed time percentage improvements for the Cosy Bonnie fastread, rewrite, and fastwrite benchmark (Compared to VAN*).*

cedure called READDIRPLUS [5]. This procedure combines READDIR and GETATTR from NFSv2: in one operation, READDIRPLUS reads the contents of a directory and returns both the entries in that directory and the attributes for each entry. The NFSv4 design took this idea a step further by creating simple *Compound Operations* [24]. An NFSv4 client can combine any number of basic NFS operations into a single compound message and send that entire message to an NFSv4 server for processing. The NFSv4 server processes each operation in the compound in turn, returning results for each operation in one reply. Aggregation of NFSv4 operations can provide performance benefits over slow network channels. In the context of system calls, the slow channels that prohibit the user application from getting optimal performance are context switches and data copies. We apply the idea of aggregation to make the slow channel more efficient, thereby improving the performance of applications.

Many Internet applications such as HTTP and FTP servers often perform a common task: read a file from disk and send it over the network to a remote client. To achieve this in user level, a program must open the file, read its data, and write it out on a socket. These actions require several context switches and data copies. To speed up this common action, several vendors created a new system call that can send a file's contents to an outgoing socket in one operation. AIX and Linux use a system call called `sendfile()` and Microsoft's IIS has a similar function named `TransmitFile()`. HTTP servers using such new system calls report performance improvements ranging from 92% to 116% [12]. `sendfile()` and similar system calls require additional effort for each new system call. Many systems also have a limit on the number of system calls that can be easily integrated into the kernel. Just as the transition from NFSv3 to NFSv4 recognized that not every conceivable compound should require a new operation, Cosy can create new compounds without the need for additional kernel modifications or many new system calls.

5.2 Zero-Copy Techniques

Zero-copy is an old concept and many ideas have been explored by researchers in different contexts. The essence of all of these attempts is to build a fast path between the user application, the kernel and the underlying device. IBM's adaptive fast path architecture [12] aims to improve the efficiency of network servers using a zero-copy path by keeping the static contents in a RAM-based cache. Zero-copy had also been used on file data to enhance the system performance by doing intelligent I/O buffer management (known as *fast buffers*) [15] and data transfer across the protection domain boundaries. The fast buffer facility combines virtual page remapping with shared virtual memory. *Tux* is a commercially available in-kernel Web server that utilizes zero-copy techniques for network and disk operations [19]. Different zero-copy techniques are useful for different applications. We studied these zero-copy techniques and adopted some of them. Cosy provides a generalized interface to utilize these zero-copy techniques.

5.3 Kernel Space Execution of Untrusted Code

Typed Assembly Language (TAL) is an approach toward safe execution of user programs in kernel mode [28]. TAL is a safe kernel mode execution mechanism. The safety is verified through the type checker, thus relying on static code checking to avoid runtime checking. Still, array bounds checking (similar to BCC [3]) is done at runtime adding overhead. In our approach, we use hardware security mechanisms such as segmentation to protect against malicious memory references [6]. This reduces runtime overhead.

Extensible operating systems like SPIN [2], ExoKernel [9, 11], and VINO [22] let an application apply certain customizations to tailor the behavior of the operating system to the needs of the application. The goal of the research in this area is to let applications extend the behavior of the system without compromising the integrity and safety of the system.

The ExoKernel allows users to describe the on-disk data structures and the methods to implement them. ExoKernels provide application specific handlers (ASHs) [30] that facilitate downloading code into the kernel to improve performance of networking applications.

SPIN allows the downloading and running of typesafe Modula-3 code. Depending upon the application SPIN can be extended by adding a new extension written in Modula-3. Extensions add special features to the existing operating system in order to enhance the performance of the application.

VINO shares a similar goal as that of the ExoKernel or SPIN. VINO allows extensions written in C or C++ to be downloaded into the kernel. VINO uses fault isolation via software to ensure the safety of the extensions [23]. It also uses a safe compiler developed at Harvard to validate memory accesses in the extension. This compiler also assures protection against self-modifying code. Cosy shares many commonalities with this work such as compiler-assisted techniques to ensure the safety of the untrusted code. Cosy, however, uses hardware-assisted fault isolation.

The problem with these approaches is their specialization: using specialized operating systems that are not widely used, or requiring languages that are not common. Conversely, Cosy is prototyped on a common operating system (Linux) and it supports a subset of a widely used language (C).

Lucco uses the software fault isolation [29] to run applications written in any language securely in the kernel. They use a binary rewriting technique to add explicit checks to verify the memory accesses and branch addresses. We provide similar guarantees but instead of using software based memory validation, we use the x86 segmentation feature to achieve the same goal. Software checks add overhead when working with extensions involving movement across multiple segments.

Proof carrying code [14] is another technique that allows the execution of untrusted code without adding runtime checks. While compiling the code, it is verified against a given policy. If the code satisfies that policy, then a proof is attached. The proof is verified quickly during runtime. For very complex code, generating a safety proof may be an undecidable task [14]; because of this, tedious hand-crafting of code may be necessary.

Packet filters also address the problem of porting user code to the kernel [10, 21]. Mogul et. al. and the BSD packet filter improve the performance of user-level network protocols by making use of a kernel resident, protocol independent packet filter. The concept of a packet filter is inherently limited to network protocols. It is useful under special circumstances; however, it is not meant to be sufficiently general to apply to all sorts of user applications. Our approach provides a more generic API which is not present in the packet filter.

Java 2 Micro Edition is designed to function as an operating system for embedded devices. Devices such as cellular phones, handhelds, and consumer electronics can download code and then safely execute it [27]. Java converts source code into an intermediate form to be interpreted by a Java Virtual Machine within a sandbox. Both Java and Cosy provide safety through runtime checking. Java, however, interprets its byte code and allows for a greater variety of extensions; Cosy simply decodes instructions passed to it from user space.

One closely-related work to ours is Riesen's use of kernel extensions to decrease the latency of user level communication [18]. The basic idea in both approaches is to move user code into the kernel and execute it in kernel mode. Riesen's proposal discusses various approaches that are adopted to address this problem. It compares various methods to achieve improved performance and then proposes to use the approach of a kernel embedded interpreter to safely introduce untrusted user-level code into the kernel. Riesen discusses the use of complier techniques to convert a C program into intermediate low-level assembly code that can be directly executed by the interpreter residing inside the kernel. Riesen's work differs from ours in that we do not interpret code to be loaded into the kernel but rather encode several calls into one structure. Unfortunately, Riesen's work was neither officially published nor completed and hence results are not available for comparison.

6 Conclusions

Our work has the following three contributions:

- We provide a generic interface to several zero-copy techniques. Thus many applications can benefit from Cosy.
- Cosy supports a subset of a widely-used language, namely C, making Cosy easy to work with. Cosy allows loops, arithmetic operations, and even function calls, thus allowing a wide range of code to be moved into the kernel.
- We have prototyped Cosy on Linux, which is a commonly-used operating system. Many widely-used user applications exist for Linux. We show improvement in the performance of such

commonly-used applications. This improvement is achieved without compromising safety.

We have prototyped the Cosy system in Linux and evaluated it under a variety of workloads. Our micro-benchmarks show that individual system calls are sped up by 40–90% for non-I/O bound common user applications. Moreover, we modified popular user applications that exhibit sequential or random access patterns (e.g., a database) to use Cosy. For non-I/O bound applications, with just very minimal code changes, we achieved a performance speedup of up to 20–80% over that of unmodified versions of these applications

6.1 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 [17]. 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 [25] and typical disk access patterns to make Cosy I/O conscious.

7 Acknowledgments

This work was partially made possible by an NSF CAREER award EIA-0133589, and HP/Intel gifts numbers 87128 and 88415.1.

The work described in this paper is Open Source Software and is available for download from `ftp://ftp.fsl.cs.sunysb.edu/pub/cosy`.

References

[1] E. W. Anderson and J. Pasquale. The performance of the container shipping i/o system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[2] 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.

[3] H. T. Brugge. The BCC home page. `http://web.inter.NL.net/hcc/Haj.Ten.Brugge`, 2001.

[4] 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.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.

[6] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 140–153, Kiawah Island Resort, near Charleston, SC, December 1999. ACM SIGOPS.

[7] R. Coker. The Bonnie home page. `www.textuality.com/bonnie`, 1996.

[8] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, June 2003.

[9] D. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[10] J. C. Mogul and R. F. Rashid and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP '87)*, November 1987.

[11] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of 16th ACM Symposium on Operating Systems Principles*, pages 52–65, October 1997.

[12] P. Joubert R. King, R. Neves, M. Russinovich, and J. Tracey. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *Proceedings of the Annual USENIX Technical Conference*, pages 175–187, June 2001.

[13] O. Krieger, M. Stumm, and R. C. Unrau. The alloc stream facility: A redesign of application-level stream i/o. *IEEE Computer*, 27(3):75–82, March 1994.

[14] G. Necula and P. Lee. Safe Kernel Extension Without Run-Time Checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, October 1996.

[15] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, December 1993.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 37–66, New Orleans, LA, February 1999. ACM SIGOPS.

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

[18] R. Riesen. Using kernel extensions to decrease the latency of user level communication primitives. `www.cs.unm.edu/~riesen/prop`, 1996.

[19] Red Hat Inc. Red Hat TUX Web Server manual. `www.redhat.com/docs/manuals/tux`, 2001.

[20] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, pages 41–54, June 2000.

[21] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Technical Conference*, pages 259–69, January 1993.

[22] 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.

[23] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–227, October 1996.

[24] 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.

[25] 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.

[26] Sun Microsystems. NFS: Network file system protocol specification. Technical Report RFC 1094, Network Working Group, March 1989.

[27] Sun Microsystems. Java 2 Platform, Micro Edition. `http://java.sun.com/j2me`, 2002.

[28] 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.

[29] 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.

[30] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proceedings of ACM SIGCOMM '96*, pages 40–52, Stanford, CA, August 1996.

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

[32] E. Zadok. *Linux NFS and Automounter Administration*. Sybex, Inc., May 2001.

[33] 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.