

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

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

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 from 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, which avoids redundant data copies.

The formation of a Cosy compound is made trivial by using a Cosy version of gcc. Cosy-gcc automatically converts user-defined code segments into compounds. To ensure 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 `/bin/grep` and `/bin/ls`. These application showed an improvement of 25–90%. 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 movement of 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 [8]. For applications like `ls`, that involve a large number of small system calls performance is hampered by the time wasted in context switching.

The general 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, 4, 5]. Applications written for Exokernels, microkernels, and SPIN have supported this idea. The common problem with these approaches is that they do not fit in the framework of current commodity operating systems. Others attempt to improve performance but at the cost of safety [6]. Software fault isolation provides a secure mechanism to execute untrusted code, but performance benefits are lost [7]. Application-specific solutions to solve the poor performance include `sendfile` and NFSv4. `Sendfile` provides a way to avoid data copies for network specific applications. NFSv4 adds

some new calls like `REaddirPLUS` to reduce the overhead due to a `readdir` followed by several `stat` calls. These approaches are successful but they are not extensible.

We present Cosy as a generic solution to safely execute the bottleneck code segment of the user application in the kernel. Cosy exploits zero-copy techniques and code aggregation to achieve maximum performance without breaking the security framework. 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 (like `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 auto 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 will convert it into a compound at compile time. Cosy-GCC also finds any data dependency 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.

Executing user code in kernel-mode requires concern for the safety of the kernel. Cosy assures safety in several ways. It invokes all the system calls the same way as a normal user process. Thus all the normal validity checks for system call invocation are performed. Cosy puts a limit on the maximum number of statements that can be executed from the kernel thereby avoiding any deadlocks or infinite loops. While executing a user-provided function, Cosy puts a limit on the execution time to avoid infinite loops in the kernel. We are currently exploring additional techniques including segmentation and bounds checking to guarantee safety.

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 without compromising security. The primary design goals were:

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 involving kernel preemption 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 burden.

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 *shared buffer* is another buffer used to facilitate zero copy within system calls that share parameters. We will look at the individual components and the internals of the compound buffer in the following subsections.

2.1.1 Kernel Extension for Cosy

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

- **cosy_init** allocates the two shared buffers that will be used by the user and kernel to exchange encoded compound and return 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 will have 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. To reduce the decoding overhead in the kernel we have limited the instructions allowed within a marked segment. This may require small rearrangement of complex code in order to make it Cosy compliant. In

the future, we plan to automatically identify these bottleneck sections. 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.

2.1.3 Cosy-Lib

Cosy-Lib provides a set of utility functions to create 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 will 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
- Loop statements
- Conditional statements
- User function pointers
- Variable and arithmetic assignments
- Labels and unconditional jumps

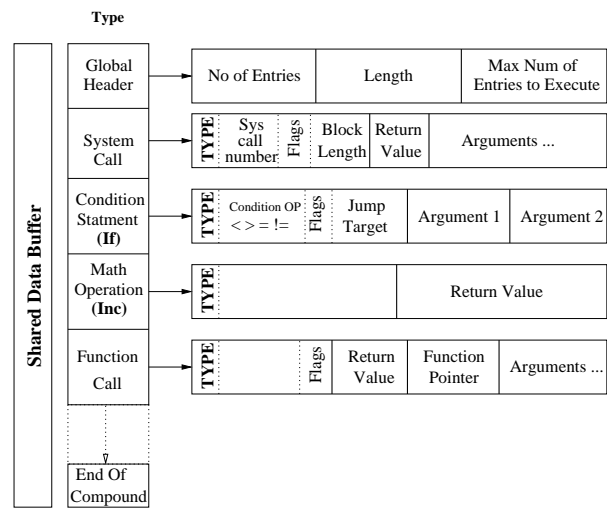


Figure 1: An Example of the Structure of a Cosy Compound

The first word of the compound buffer is the global header. It tells the total number of entries in the compound and the length of the compound in terms of total number of words. The field, “maximum number of entries to execute”, puts an upper bound on the total instructions that can be executed. This entry ensures protection against unending compounds. The rest of the compound contains a set of entries of the form “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 will contain system call number and flags. The flags indicate whether the argument is the actual value or it is a reference to the output of some other 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 case of system call, math operation, function call) one position is reserved to store the result of the entry. 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 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 necessary 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. 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 upon the type of application, different zero copy techniques will be 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 shared 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 safety in the kernel. It tries to make efficient use of the hardware and software checks provided by the underlying architecture and the operating system. Static checks are enforced by Cosy-GCC. Necessary dynamic checks are performed by the Cosy kernel module. While compiling the user marked bottleneck code segment, Cosy-GCC makes sure that there are no unsupported or ambiguous statements in the marked segment. The system call invocation by the Cosy kernel module is the same as a normal process and hence all the necessary checks are observed. The Cosy kernel counts the total number of entries decoded and executed at any time. When this number exceeds the maximum entry value, it returns with an appropriate error code. Putting the limit on the total number of entries is not always sufficient. As the user supplied code can have infinite loops that may cause kernel to hang. Cosy provides a secure mechanism to detect such hung processes and terminate them using a preemptible Linux kernel.

We will use a combination of techniques, both static and dynamic to provide maximum protection for the kernel. Our aim is not to be able to execute any random code kernel mode, but rather we want to provide a secure mechanism so that any valid code can be safely executed in the kernel.

We are currently exploring two techniques for safe kernel-mode code execution: segmentation [3] and bounds-checking [2]. In some specific situations a bounds-checking compiler is useful to detect memory violations without much overhead. We also plan to use similar techniques to provide protection against typical memory violations.

We will use compiler techniques to ensure that the untrusted code does not involve any privileged instruction, and to protect against self-modifying code.

We exploit the x86 segmentation feature to isolate untrusted code. We put the untrusted code into a different segment, but one that executes at the same privilege as the kernel. To protect the mainline kernel we mark the kernel segments read only while executing the untrusted code.

Currently we are limiting the type of instructions that can be executed in the kernel (e.g., no privileged instructions, no self-modifying code). Our aim is to minimize such limitations so that the largest variety of code can be executed in kernel mode without affecting safety.

2.4 Cosy Example

In this section we will illustrate some simple examples of code using Cosy. We will write them once using

Cosy-GCC and without using Cosy-GCC. To improve clarity and conserve space we do not include any error checking. We assume that the system calls without any error checking will always succeed.

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

```

1 cosy_init();
2
3 COSY_START();
4 ifd = open(ifile, O_RDONLY);
5 ofd = open(ofile, O_WRONLY);
6
7 do {
8     rlen = read(ifd, buf, 4096);
9     wlen = write(ofd, buf, rlen);
10 } while(wlen == 4096);
11 COSY_END();
12 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 will take 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 will release 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 following 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.
3. **COSY-ZERO:** This configuration makes use of compounds and the zero copy system calls provided by Cosy.

We performed our tests on a Intel Pentium-IV 1.7 GHz machine with 64MB of RAM and a 7200 RPM 20GB ATA/100 hard-drive. We repeated the tests 20 times to verify our results. The observed standard deviations were less than 5%. We report results of two benchmarks, `ls` and `grep`, to evaluate the performance of Cosy.

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

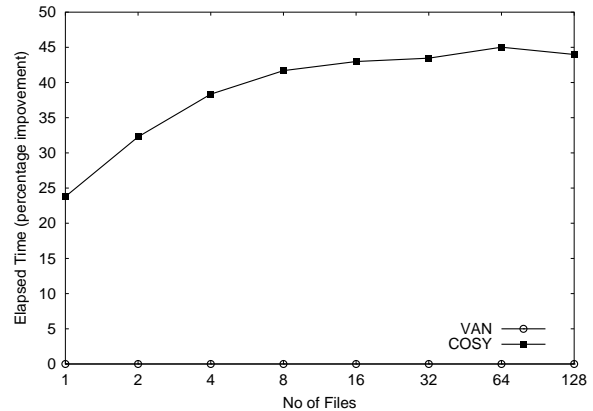


Figure 2: Elapsed time % improvement for `ls -l`

The Cosy version of `ls -l` performs consistently better than the generic `ls -l`, with a performance increase of almost 45% as shown in Figure 2. The improvement increases linearly with the number of files. The decrease in the percentage improvement occurs at 128 files because the cost of evaluating the compound inside the kernel becomes more significant with an increasing number of files.

grep To find out the effect of Cosy on data intensive applications we instrumented `grep`. This benchmarks 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 configuration mentioned at the start of this section.

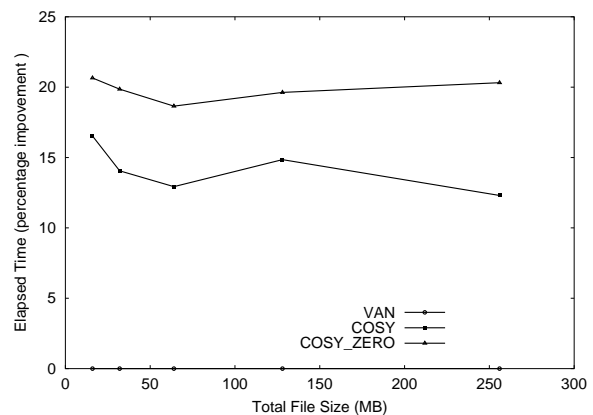


Figure 3: Elapsed time % improvement for `grep`

The Cosy version of `grep` with zero copy, performs 20% better than the normal version (see figure 3). The nonzero copy version also shows improvement of 15%. 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 benefits.

4 Conclusions

In our work we introduce a safe yet efficient mechanism to execute bottleneck code segments in a user application, in kernel mode. We exploit 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. Our benchmarks prove the usefulness and effectiveness of compound system calls. We show a speed improvement of 45–90% depending upon the type of application.

We provide various features to ensure safe execution of the compound in the kernel. We use kernel preemption and compiler techniques to ensure safety. We follow normal system call checks while processing a compound thereby avoiding any accidental security problems. We are exploring hardware features like x86 segmentation, bound-checking compilers, and sandboxing techniques to make this security mechanism more robust. Our current work aims at finding out the optimum point in the safety, performance, and isolation triangle.

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] H. T. Brugge. The BCC home page. <http://web.inter.NL.net/hcc/Haj.Ten.Brugge>, 2001.
- [3] 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.
- [4] 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.
- [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] T. Maeda. Safe Execution of User programs in kernel using Typed Assembly language. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/tosh_master_kml_e.pdf, 2002.
- [7] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.
- [8] 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.