Efficient and Safe Execution of User-Level Code in the Kernel

Erez Zadok, Sean Callanan, Abhishek Rai, Gopalan Sivathanu, and Avishay Traeger
Stony Brook University

Appears in the proceedings of the 2005 NSF Next Generation Software Workshop, in conjunction with the 2005 International Parallel and Distributed Processing Symposium (IPDPS 2005)

Abstract

This project has two goals. The first goal is to improve application performance by reducing context switches and data copies. We do this by either running select sections of the application in kernel-mode, or by creating new, more efficient system calls. The second goal is to ensure that kernel safety is not violated when running user-level code in the kernel. We do this by implementing various hardware- and software-based techniques for runtime monitoring of memory buffers, pointers, as well as higher-level, OS-specific constructs such as spin-locks and reference counters; the latter techniques can also be used for code written specifically for the OS. We prototyped several of these techniques. For certain applications, we demonstrate performance improvements as high as 80%. Moreover, our kernel safety checks show overheads that are as little as 2%.

1 Introduction

Software development is hard, and kernel code development is harder, which is why most programs are written in user space, where more tools are available to develop and debug one's code. Developers, however, would like to develop and run applications in the kernel because they can perform better there: overheads due to context switching and data copies between the kernel and the user address spaces are eliminated. Kernel debuggers, even when they are available, are unsuitable for debugging race conditions and other timing-related issues.

Our goal in this project is to combine the best of both worlds: allow users to develop and debug their code in user-level, while running their code inside the kernel. To achieve that, we are developing a kernel-aware C development environment (KGCC), kernel infrastructure to work with KGCC-compiled code, as well as a host of useful utilities. Our compilation system has both static and dynamic components. To improve application performance, we use two approaches. First, we mark the code to instruct the compiler to pack code segments at run time into a single small buffer, make it available for the kernel using specially-mapped physical memory, and then instruct the kernel to execute the code segment directly in kernel-mode. Second, we captured system-call traces for many commodity user programs such as graphical environments, Web browsers, long-running daemons (e.g., Sendmail and Apache), and even small programs like `/bin/ls`. We analyzed these traces and located a number of frequently-executed sequences for which a new, unified system call would be more efficient. We implemented and benchmarked some of these sequences.

Running user-written code inside the OS requires that the safety of the kernel environment—and hence of all users and applications—not be violated. Therefore, a substantial part of our research focuses on techniques to make kernel code safer to run. The techniques we are developing are applicable not just for user-level code that is executed in the kernel, but also for (1) code written directly for the kernel, and (2) long-running user level programs that provide important privileged services that must run as reliably as possible (e.g., e-commerce applications). To improve kernel safety, we use three approaches which we are integrating into KGCC. First, we used hardware-based techniques such as segmentation. We used parts of the user-level bounds-protecting library Electric Fence [12] in the Linux kernel. Electric Fence protects buffers from overflow and underflow by placing the buffers on page boundaries and inserting a protected "guardian" page at the boundary; any access to a guardian page results in a hardware page-fault. Second, we used software-based techniques. We ported the GNU Bounds-Checking Compiler (BCC) [5] to the Linux kernel. BCC includes not just bounds-checking tests, but also pointer checking and validation, malloc/free checking, and a few more. Third, we are enhancing KGCC to include OS-specific tests. Our KGCC can check and validate at run time the proper use of kernel-specific aspects such as semaphores and spin-locks, reference counts, and other higher-level invariants.

Instrumenting a lot of kernel code could add overhead. We are considering some techniques for turning off certain checks using simple heuristics such as the number of times a particular check was executed. Eventually, we hope to develop tools and techniques that will make kernel code development as easy as user-level code, while making both future operating systems and user-level applications more efficient and safe.

This paper is organized as follows. Section 2 discusses techniques for speeding applications up when they invoke system calls. Section 3 presents techniques for verifying kernel code safety. Section 4 concludes.

2 System Call Optimizations

Long-running server applications can easily execute billions of common data-intensive system calls each day. In exchange for providing a secure and uniform interface for user applications to request services from the kernel, system calls incur large overheads due to data copies across the user-kernel boundary and context switches. We lessen these penalties in two ways. First, we analyze system call invocations in a system, extract commonly-used sequences, and develop new system calls to perform the task of a given sequence. This reduces both context switches and data copies. We implemented several of these system calls and benchmarked them to estimate the benefits of this technique. Second, we introduce a new framework, *Compound System Calls* (Cosy), to enhance the performance of such applications. Cosy provides a mechanism to execute data-intensive code segments in the kernel safely. 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 and instrumented a few data-intensive applications such as those with database access patterns. Our benchmarks show performance improvements of 20–80% for CPU-bound applications.

2.1 Related Work

The networking community has long known that better throughput can be achieved by exchanging more data at once than repeatedly in smaller units. For example, a large number of READDIR operations are followed by many GETATTR operations. *Compound Operations* were introduced in NFSv4 [18], in which a client may encapsulate several operations for processing. This aggregation 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.

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 speed up this common action, AIX and Linux created a system call called `sendfile` and Microsoft's IIS has a similar `TransmitFile` function. HTTP servers using these system calls report performance improvements ranging from 92% to 116% [7].

The Cassyopia project [14] suggested to profile the address-space–crossing behavior of a component (between address spaces on the same or different machines), and then have the compiler reduce the number of crossings in order to improve performance.

Zero-copy is a well known technique to share the data instead of copying. IBM's adaptive fast path architecture [7] aims to improve the efficiency of network servers. Zero-copy is also used on file data to enhance the system performance by doing intelligent I/O buffer management (known as *fast buffers*) [3].

Extensible operating systems let an application apply certain customizations to tailor the behavior of the OS to the needs of the application. The ExoKernel allows users to describe the on-disk data structures and the methods to implement them. ExoKernels provide application-specific handlers [22] that facilitate downloading code into the kernel to improve performance of networking applications. SPIN allows type-safe Modula-3 code to be downloaded and run in the kernel [1]. VINO allows extensions written in C or C++ to be downloaded into the kernel. It uses fault isolation via software to ensure the safety of the extensions [17]. It also uses a safe compiler to validate memory accesses in the extension and assure protection against self-modifying code. Riesen discusses the use of compiler techniques to convert a C program into intermediate low-level assembly code that can be directly executed by an interpreter inside the kernel [15]. Unfortunately, the work was neither officially published nor completed, and results are not available for comparison.

2.2 System Call Consolidation

Design The first step in finding system call patterns was to collect logs of system calls. This was done using a combination of `strace` and the system call auditing support in Linux 2.6. Once the system call activity was logged, we used a script to create a system call graph [14] and searched for patterns. This is a weighted directed graph with vertices representing system calls and an edge between V_1 and V_2 having a weight equal to the number of times system call V_2 was invoked after V_1. Paths with large weights are likely to be good candidates for consolidation.

We found several promising system call patterns, including `open-read-close`, `open-write-close`, open-fstat, and readdir-stat. We implemented several new system calls to measure the improvements. The main savings for the first three combinations would be the reduced number of context switches. The readdirplus system call returns the names and status information for all of the files in a directory. This combines readdir with multiple stat calls. Here we save on both context switches and data copies, because once we get the file names we can directly use them to get the stat information. This is a well-known optimization, and was introduced in NFSv3 [2].

Evaluation We tested readdirplus on a 1.7GHz Intel Pentium 4 machine with 884MB of RAM running Linux 2.6.10. We used an IDE disk formatted with an Ext3 file system. We benchmarked readdirplus against a program which did a readdir followed by stat calls for each file. We increased the number of files by powers of 10 from 10 to 100,000 and found that the improvements were fairly consistent: elapsed, system, and user times improved 60.6–63.8%, 55.7–59.3%, and 82.8–84.0%, respectively.

To see how this might affect an average user's workload, we logged the system calls on a system under average interactive user load for approximately 15 minutes. We then calculated the expected savings if readdirplus were used. The total amount of data transfered between user and kernel space was 51,807,520 bytes, and we estimate that if readdirplus were used we would only transfer 32,250,041 bytes. We would also do far fewer system calls—17,251 instead of 171,975. This would translate to a savings of about 28.15 seconds per hour. Although this savings is small, it is for an interactive workload. We expect that other CPU-bound workloads, such as mail and Web servers, would benefit more significantly from new system calls.

2.3 Compound System Calls

Design Often only a critical portion of the application code suffers due to data movement across the user-kernel boundary. *Compound System Calls* (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 [13].

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*. The Cosy system uses two buffers for exchanging information. The first is a *compound buffer*, where the compound is encoded. The buffer is shared between the user and kernel space, so the operations that are added by the user into the compound are directly available to the Cosy Kernel Extension without any data copies. The second is a *shared buffer* to facilitate zero-copying of data within system calls and between user applications and the kernel.

The first component, 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, and determines if the input parameter of the operations is the output of any of the previous operations. We limited Cosy to the execution of only a subset of C in the kernel. One of the main reasons is safety. Another concern 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 second component of Cosy, Cosy-Lib, 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. The functioning of Cosy-Lib and the internal structure of the compound buffer are entirely transparent to the user. The final component is the Cosy kernel extension, which is the heart of the Cosy framework. It decodes each operation within a compound and then executes each operation in turn.

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 uses hardware and software checks provided by the underlying architecture and the operating system to do this efficiently. Two of the safety features are a preemptive kernel to avoid infinite loops and x86 segmentation to protect kernel memory.

To remove the possibility of infinite loops in the kernel, we use a preemptive kernel that checks the running time of a Cosy process inside the kernel every time it is scheduled out. If this time has exceeded the maximum allowed kernel time then the process is terminated.

Cosy supports two approaches to protect kernel memory. 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. The second approach uses a combination of static and dynamic methods to ensure safety. 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. 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 is also vulnerable to hand-crafted user functions that are not compiled using Cosy-GCC.

Evaluation We have prototyped the Cosy system in Linux and evaluated it under a variety of workloads [13]. Our micro-benchmarks show that individual system calls are sped up by 40–90% for common CPU-bound user applications. Moreover, we modified popular user applications that exhibit sequential or random access patterns (e.g., a database) to use Cosy. For CPU bound applications, with very minimal code changes, we achieved a performance speedup of up to 20–80% over that of unmodified versions of these applications.

2.4 Status and Future Work

In the future, we would like to modify Cosy to automate the job of deciding which code should be moved to the kernel using profiling. In addition, we would like to create the option of executing 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 time 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.

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 [19] and typical disk access patterns to make Cosy I/O conscious.

We will continue to analyze system call patterns on machines being used for various purposes, and implement new system call suites that cater to their workloads. This way, an administrator can choose to use those system calls which are tailored to applications such as mail servers or Web servers.

3 Runtime Validation of Kernel Code

Programmers make mistakes, and C semantics leave ample opportunity for introducing memory errors. This problem becomes all the more acute when programming inside the kernel as a small memory-access bug could crash the entire system. Runtime bounds checking through hardware is an efficient method of detecting program bugs. We have developed such a system called *Kefence*. KGCC, which is a software based approach provides more comprehensive checking.

While runtime bounds-checking is an effective technique to verify properties specific to C language semantics, kernel programmers frequently want to verify higher-level properties stated in terms of program semantics rather than language semantics. In the kernel, there are many properties we would like to verify: spinlocks that are locked are later unlocked, reference counters are incremented and decremented symmetrically, interrupts that are disabled are later re-enabled. To allow for checking of these, we have developed an event monitoring infrastructure with support for on-line analysis in the kernel and in user space, as well as logging for later analysis.

3.1 Related Work

Validation of operating systems has historically been performed using techniques in the following areas: static checking, verification, compiler assistance, and external runtime testing.

Dynamic checking, as employed in Bounds-Checking GCC (BCC) [5] and other systems, is an example of compiler–assisted verification. In this case, it is used to insert runtime checks for memory corruption such as buffer overflows in C [12].

Electric Fence uses a system's virtual memory hardware to place an inaccessible memory page immediately after (or, at the user's option, before) each allocated memory area. When software reads to or writes from this page, the hardware issues a segmentation fault, stopping the program at the offending instruction. It is then easy to find the statement in the C source using a debugger. Memory that has been released by `free` is made similarly inaccessible.

Wahbe et al. use software fault isolation [21] to run applications written in any language securely in the kernel. They use binary rewriting to add explicit checks to verify the memory accesses and branch addresses.

Tracing, a special case of event monitoring, has been extensively used to analyze the behavior of applications as they interact with system APIs. Tracing involves recording events in a log file, and using that file for later analysis. This has been done for file systems [10] and network applications [8].

Profiling has been used to find bottlenecks in application performance. Counters keep track of how frequently applications' basic blocks are executed. A derivative of this technique, kernel profiling, is a form of on-line monitoring which gathers unified statistics about system behavior [20]. Profiling based on logs has also been used to monitor system performance in distributed applications [4].

3.2 Kefence

Design Kefence is designed to detect memory buffer overflows at the hardware level. Kefence aligns memory buffers allocated in the kernel virtual address space (using `vmalloc`) to page boundaries. The kernel's `vmalloc` function allocates one or several pages for each request, facilitating this alignment. A guardian page table entry (PTE) is added adjacent to each buffer so that whenever a buffer overflow occurs, the guardian PTE is accessed. The guardian PTE has read and write permissions disabled; hence, accessing it causes a page fault. The page fault handler of the Linux kernel is modified so that whenever there is an access to a guardian PTE, it reports a buffer overflow.

Exact details about the context and location of buffer overflows are logged through syslog. The modified page fault handler can be configured to perform various additional tasks. When security is critical, Kefence can be configured to crash the module upon a memory overflow, thereby preventing further malicious operations. The system administrator can look at the logs to determine the location of overflow. If debugging is more important, Kefence can be configured to just log the buffer overflow without terminating the module. We implemented this by auto-mapping a read-only or read-write page to the guardian PTE whenever there is an overflow. This way the code which caused the overflow can either be allowed to write or to just read the out-of-bounds memory locations. Since the logs contain full information about the location and the code which caused the overflow, buffer overflows in kernel code can be diagnosed easily. Kefence can do this in real time, making it suitable for security critical applications.

Since Kefence can only protect virtually-mapped buffers, those allocated using `kmalloc` are not protected. Therefore, to add bounds checking to a kernel module, one must use `vmalloc` instead of `kmalloc` for memory allocations. We have modified Linux header files in such a way that this replacement is done automatically if a special compiler flag is set.

Using `vmalloc` consumes more virtual memory, since it allocates at least a page for each memory allocation. This is partly mitigated by the fact that modern 64-bit architectures make the address space a virtually inexhaustible resource. However, replacement of `kmalloc`s with `vmalloc`s results in extra consumption of physical memory because the memory is allocated in units of pages. To speed up the default `vfree` function we have added a hash table to store the information about virtual memory buffers. Since the alignment of buffers to page boundaries can be done either at the beginning or at the end, Kefence cannot detect buffer overflows and underflows simultaneously, unless the allocation is in multiples of the page size. While this is a common case inside the kernel, we have found overflow protection to be sufficient in most other cases.

Evaluation To evaluate the performance of Kefence, we instrumented a stackable file system [23] called Wrapfs. Wrapfs is a wrapper file system that just redirects file system calls to a lower-level file system. The vanilla Wrapfs uses `kmalloc` for allocation. Each Wrapfs object (inode, file, etc.) contains a private data field which gets dynamically allocated. In addition to this, temporary page buffers and strings containing file names are also allocated dynamically. We mounted Wrapfs on top of a regular Ext2 file system with a 7,200 RPM IDE disk to conduct our tests. In the instrumented version of Wrapfs, we used `vmalloc` for all memory allocations so that we could exercise Kefence for all dynamically allocated buffers.

We compiled the Am-utils [11] package over Wrapfs and compared the time overhead of the instrumented version of Wrapfs with vanilla Wrapfs. The instrumented version of Wrapfs had an overhead of 1.4% elapsed time over normal Wrapfs. This overhead is due to two main causes. First, `vmalloc`/`vfree` functions are slower than `kmalloc`/`kfree` functions. Second, allocating an entire page for each memory buffer increases TLB contention. We found that the maximum number of outstanding allocated pages during the compilation of Am-utils over the instrumented version of Wrapfs was 2,085 and the average size of each memory allocation was 80 bytes. We conclude that Kefence performs well for normal user workloads. However, memory-intensive code may exhaust virtual or physical memory.

3.3 Event Monitoring

Design When designing the event monitoring framework, we wanted it to be flexible enough to allow for instrumentation of any part of the kernel. This dictated that it should have two main properties: generality and performance sensitivity. For generality, the framework should make as few assumptions as possible about the events that are being monitored. This should be reflected both in a simple and generic API, and also in the non-intrusiveness of the code (e.g., the code should not block because some portions of the kernel cannot be preempted). For performance sensitivity, the programmer ought to be able to insert checks for frequent events directly into the kernel, as well as being able to monitor infrequently-occurring events easily in user space.

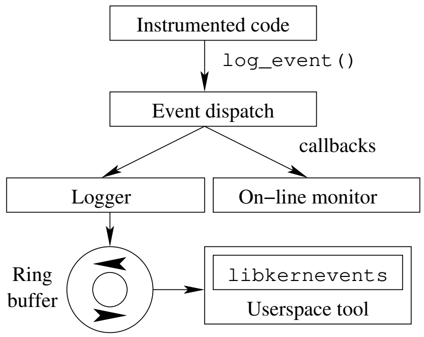

Figure 1: Event monitor structure

Figure 1 shows the event monitor's internal structure. The `log_event` call invokes an event dispatcher, which in turn invokes a set of callbacks. When high performance is needed, an event monitor should be developed as a kernel module and register a callback with the dispatcher.

Kernel-space on-line event monitors are invoked synchronously via their callbacks; user-space event monitors receive events through a character device interface to a lock-free ring buffer. Because the ring buffer is lock-free, we can instrument code that is invoked during interrupt handlers without fear that the interrupt handler will block, which makes the code more general. We have been able to instrument scheduler and interrupt handler code safely using this module. User-space applications can link with `libkernevents` to copy log entries in bulk from the kernel and then read them one by one.

Each event is recorded by a structure that contains a `void *` that references the object affected by the event (e.g., this field can be used to extract the current value of a reference counter); an integer that encodes the type of event (e.g., increment or decrement for a reference counter); and the source file and line number that triggered the event. This structure has been designed to minimize the size of individual log entries while providing sufficient generality to allow most pertinent information to be recorded.

Evaluation We tested our framework under the PostMark benchmark. The test system was a P4 1.7GHz CPU with 884MB RAM and a Western Digital Caviar 20GB hard drive, under Linux 2.6.9. For the logging test (see below) we used a Quantum Atlas 15K SCSI drive on an Adaptec 39160D SCSI adapter to hold log data.

We ran the control test on a vanilla 2.6.9 kernel. We then added instrumentation for the dentry cache lock, `dcache_lock`, which prevents race conditions in file-system name-space operations such as renames. During our benchmark, this lock was hit an average of 8,805 times a second; the benchmark ran for an average of 85.4 seconds. Adding the event dispatcher and ring buffer resulted in a 3.9% overhead; running a user-space logger built around `librefcounts` in parallel with PostMark increased the overhead to 103%.

Running a user-space program that acts like the logger but does not write to disk still gave a 61% overhead, and system time was effectively constant for all runs, regardless of instrumentation. Hence the inefficiencies did not arise from the kernel infrastructure. We believe that the overhead from the user-space logger is due to inefficiencies in the user-kernel interface; in our current prototype, `librefcounts` polls the character device continuously rather than using blocking reads.

3.4 KGCC

KGCC is a compilation framework that allows instrumentation of kernel code. This has many uses: it can be used to perform runtime bounds checking, to monitor object reference counts, and to profile kernel code. KGCC is derived from Jones and Kelly's bounds checking patch to GCC, called BCC [5]. However, KGCC extends on the functionality of BCC in several ways and has broader scope. Because KGCC is based on GCC, it can leverage GCC's optimization and analysis features.

BCC BCC inserts checks into the code at compile time to perform runtime bounds checking. All operations that can potentially cause bounds violations, like pointer arithmetic, string operations, memory copying, etc. are preceded by checks. The checks are simply function calls to the BCC runtime environment, which maintains a map of currently allocated memory in a *splay tree*; the tree is consulted before any memory operation. However, BCC fails to compile a majority of open-source C programs [16]. This is partly due to bugs in BCC and partly because it gets confused by the complicated coding style in these programs. While working towards a kernel-ready BCC, we also created a more robust version of BCC, fixing several serious bugs (e.g., several problems with inline function calls and concurrency).

Out-of-bounds pointers One of the problems with BCC is its mishandling of temporary *out-of-bounds pointers*. C programs often generate temporary addresses that are invalid but are needed as part of a larger computation. For example, in the expression `ptr+i-j`, where `ptr` is a pointer, and `i` and `j` are integers, it is possible for `ptr+i` to be outside the memory area of the object pointed to by `ptr` even though the whole expression on evaluation does translate to a valid address. BCC flags an error for such temporary out-of-bounds addresses, which is undesirable.

Though replacing invalid addresses with references to external table entries suppresses some errors and is one possible solution [16], it introduces problems when the replacement is passed to code that was not compiled with BCC. Our solution is based on the concept of *peers*. Whenever an out-of-bounds address is created by arithmetic on an object O, we insert a special out-of-bounds (OOB) object at the new address into the address map, and make it a peer of object O. Our KGCC runtime permits only pointer arithmetic on OOB objects, which can either generate another peer or return to O's bounds. Our approach handles out-of-bounds pointers without suffering from the problems of the replacement-based approach.

Porting BCC to the kernel As we modified BCC to support the Linux kernel, we dealt with several issues relating to symbols and the semantics of the kernel's C runtime environment and segmentation. We also made modifications to the kernel and module initialization codes, and added support for multi-threading and stack management.

KGCC performance Instrumented kernel components suffer from two problems. First, instrumentation imposes memory overhead, which is undesirable since kernel memory is frequently direct-mapped. A program fully compiled with all the default checks in BCC could be up to 15 to 20 times larger than when compiled with GCC. While the BCC runtime adds a small fixed-size overhead, the bulk of the additional code is from thousands of individual checks. Second, instrumented code usually runs slower, because it must execute additional instructions to track and validate accesses. KGCC handles these two problems in a uniform way.

During compilation, KGCC employs heuristics to eliminate unnecessary checks. For example, KGCC does not check stack objects whose addresses are not taken at any point in the code. The CCured project employs a more comprehensive approach to remove unnecessary checks [9]; we plan to integrate it with KGCC. Another technique, *common subexpression elimination*, allowed us to reduce the number of checks inserted by more than half for typical kernel code. KGCC also employs other optimizations like argument packing for checking of function calls, mechanisms to reduce and localize instruction cache footprint, and trading space for performance by caching more state in the KGCC library in order to reduce communication with the module.

Evaluation We compared the performance of a KGCC-compiled Reiserfs module to a vanilla GCC-compiled module on Linux 2.6.7. We ran a CPU-intensive benchmark, an Am-utils compile. The system time for KGCC-compiled Reiserfs was 33% greater than vanilla GCC, while the elapsed time was 20% greater. We also ran the I/O-intensive benchmark PostMark [6]. In this case, the system time was 14 times greater for KGCC-compiled Reiserfs while the elapsed time was 3 times greater.

3.5 Status and Future Work

Kefence We currently have a working implementation of Kefence for the Intel 386 platform. Because converting all `kmalloc` calls to `vmalloc` calls consumes more memory, we are investigating methods to dynamically decide which memory should be protected at runtime.

Event monitoring We used our event monitor to instrument a variety of kernel components with acceptable overheads. We intend to develop on-line, in-kernel monitors for reference counters, spinlocks, and semaphores, as well as tools that allow for more in-depth analysis of performance bottlenecks related to these objects.

KGCC KGCC currently stores the address map of allocated objects in a splay tree, which brings the most recently accessed node to the top during each operation. This results in nearly optimal performance when there is reference locality. However, when multiple threads make use of the same splay tree, the splay tree is no longer as efficient, because different threads have less locality. We are currently investigating data structures better suited for multi-threaded code.

There are two techniques we intend to apply to the entire KGCC framework: selective instrumentation and dynamic deinstrumentation. First, we intend to make the compiler capable of inserting instrumentation based on rules such as "instrument every operation on an `inode`'s reference count." Second, as code paths execute safely more times and more often, one can state with greater confidence that they are correct. We intend to implement instrumentation that can be deactivated when it has executed a sufficient number of times, reclaiming performance quickly as the confidence level for frequently-executed code becomes acceptable.

As part of the implementation of these techniques, we plan to research and develop several two new technologies for integration into KGCC. First, we plan to develop a language that specifies code patterns that the KGCC compiler can then recognize and instrument, in the spirit of aspect-oriented programming. Expressions in this language would match patterns in a parsed and type-checked version of the kernel's source code. This would eliminate tedious and error-prone manual instrumentation. Second, we plan to develop a means for direct, code-level modification of an executable, like the Linux kernel, at run-time. A binary would be augmented with its parse tree and compiler-level intermediate representation (IR). The IR would contain pointers into the binary's text segment, which would be updated as the binary is converted into an executing image at runtime.

New code could be inserted by using the existing parse tree and symbol tables to convert it to IR, then compiling that IR to binary code and modifying the appropriate sections of the program's text segment.

4 Conclusions

We have shown that significant performance improvements of user applications are possible by reducing the number of context switches and data copies needed. Additionally, we have provided tools for ensuring that these complex operations, and indeed any code inserted into the Linux kernel, are safe.

Our work on system call consolidation has provided secure, efficient replacements for commonly-used sequences of system calls, improving their performance by up to 63%. For operations that are performance-critical but application-specific, we have provided a mechanism, Cosy, to allow an application to load and run code directly in the kernel, accelerating them by up to 90%.

In terms of safety, our work on Kefence combats memory errors, disabling code before it overwrites other kernel data. The KGCC compiler embeds checks into kernel code, ensuring that no pointers are dereferenced if they point outside safe areas. Finally, our event monitoring framework allows programmers to verify that kernel code adheres to high-level safety rules in its interactions with kernel objects like locks and reference counts. We are currently devising mechanisms to disable checks when they have executed enough times.

Our research has made it easier to run user-space code in the kernel, and to monitor it for safety. As our research progresses, we believe that the distinction between user and kernel code, both in terms of performance and ease of development, will gradually disappear.

5 Acknowledgments

This work was made possible by NSF CAREER award CNS-0133589, and HP/Intel gifts 87128/88415.1. Special thanks go to other contributors to this project: Alexander Butler, Mohan-Krishna Channa-Reddy, Salil Gokhale, Nikolai Joukov, Aditya Kashyap, Devaki Kulkarni, Amit Purohit, Joseph Spadavecchia, and Charles P. Wright.

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 *Proc. of the 15th ACM SOSP*, pages 267–284, Copper Mountain Resort, CO, December 1995.

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

[3] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM SOSP*, Asheville, NC, December 1993.

[4] M. Gardner, W. Feng, M. Broxtony, A. Engelhart, and G. Hurwitz. MAGNET: A tool for debugging, analyzing and adapting computing systems. In *Proc. of the 3rd CCGrid*, pages 310–317, 2003.

[5] R. Jones and P. Kelly. Bounds Checking for C. Technical report. `www-ala.doc.ic.ac.uk/phjk/BoundsChecking.html`.

[6] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

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

[8] J. Mogul. Observing TCP dynamics in real networks. In *Proc. of the Conference on Communications Architectures and Protocols*, pages 305–317, 1992.

[9] G. C. Necula, S. McPeak, and W. Weimer. Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, January 2002.

[10] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. of the 10th SOSP*, pages 15–24, Orcas Island, WA, December 1985.

[11] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[12] B. Perens. *efence(3)*, April 1993.

[13] A. Purohit. A System for Improving Application Performance Through System Call Composition. Master's thesis, Stony Brook University, June 2003. Technical Report FSL-03-01, `www.fsl.cs.sunysb.edu/docs/amit-ms-thesis/cosy.pdf`.

[14] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Cassyopia: Compiler Assisted System Optimization. In *Proc. of the 2003 HotOS IX*, Lihue, HI, May 2003.

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

[16] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proc. of the NDSS Symposium*, pages 159–169, February 2004.

[17] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the 2nd OSDI*, pages 213–227, Seattle, WA, October 1996.

[18] 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.

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

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

[21] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient Software-Based Fault Isolation. In *Proc. of the 14th SOSP*, pages 203–216, Asheville, NC, December 1993.

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

[23] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.