Linux Kernel Specialization for Scientific Application Performance
UCSB Technical Report CS2005-29 *

Lamia M. Youseff Rich Wolski Chandra Krintz

Department of Computer Science
University of California, Santa Barbara
{lyouseff, rich, ckrintz}@cs.ucsb.edu

Abstract

Linux has emerged as the system-of-choice in academic and production scientific computing settings. Much of the functionality in the Linux kernel, however, has been designed to arbitrate between competing applications thus ensuring safety and fairness. In scientific computing settings, particularly where resources are space-shared and "batch" scheduled, an executing application has exclusive access to the resources it is using. As, much of the Linux mechanism devoted to the arbitration of sharing is unneeded, and any performance degradation it introduces is purely perceived as overhead.

In this paper, we investigate the performance effects of specializing the Linux kernel for the dedicated execution of scientific applications. We analyze the kernel execution behavior of a publicly available weather modeling application and propose two different specializations that are designed to improve its performance. We are able to achieve a maximum performance improvement of 24% *over a "stock" Linux installation though our techniques, thus illustrating the potential power of this approach.*

* This work is sponsored in part by a grants from the National Science Foundation (ST-HEC-0444412), Intel Corporation, and Microsoft Research.

1 Introduction

Clusters of workstation-class computers have emerged as popular, cost-effective, and high-performance platforms for the execution of the next generation of scientific applications. Low per-unit cost, advances in computing and communication power, and the availability of Linux as a free, easy-to-use, and nearly standard operating system, make high-end computing with these systems accessible both to a very large developer base and to a wide range of users.

This accessibility is due primarily to the ubiquity Linux and its extensive support for a wide range of devices, services, and systems. System administrators commonly employ Linux *both* for locally-controlled, highly responsive application development environments and for batch-controlled, production clusters for repeated, large-scale execution. As a result, scientific programmers develop, debug, and tune their programs and then submit them to a Linux-based, high-end cluster with little or no porting effort.

A key limitation to the use of Linux for high-end cluster computing however, is its potential performance impact on application execution. Linux, like other general-purpose operating systems (OSs) with commercial application, continues to evolve to support an enormous range of user requirements and preferences, application

domains, and devices (including supercomputers, web-servers, hand-helds, and cellular phones).

As a consequence of this tension between application requirements, the Linux OS includes many features and built-in policies that do not promote the performance of high-end scientific applications. In addition, scientific applications executing in clustered settings are frequently large, resource intensive, long-running, and use space-sharing to gain exclusive access to the machines they use through a batch system. They do not compete dynamically for processor and I/O resources since the batch system ensures that any processors allocated to an application are not time-shared by other applications. None the less, the portability that Linux affords combined with the familiarity that its wide-spread popularity has bred make it a de facto standard operating system for clustered architectures.

The goal of our research is to identify and develop techniques that maintain the ease-of-use and cost benefits of Linux while enhancing the performance achievable by high-end scientific applications executing in large-scale cluster computing settings. In particular, we are exploring ways to exploit the exclusive processor access that batch scheduling implements to relax or eliminate unneeded mechanisms that are designed to facilitate effective time-sharing, but which introduce unnecessary overhead in a space-sharing context. With this paper, we take an initial step in this direction. We present results detailing the effect of two different specialization approaches on a scientific application, and also on an exemplar benchmark we developed based on inspection of the I/O patterns in that application.

Surprisingly, we find that each approach has a different effect, and also that the effect vary by Linux kernel version. Moreover, from the analysis, the source of this variation is unclear. We believe that, as a result, a profile-based approach will be warranted as a simple static analysis yields conflicting results.

In the next section, we motivate and overview our research approach for customizing Linux for scientific applications. We then present a behavior analysis (Section 3) for two programs (one hand-coded benchmark and one scientific application) and articulate how we specialize the Linux I/O subsystem for each (Section 4). In the same section, we also present the performance impact of our techniques and discussion of the results. We present related work in Section 5 and our conclusions and future research plans in Section 6.

2 Application Specific Linux

The focus of our research is to enable application-specific Linux customization for scientific programs in an effort to enable significantly higher application performance while maintaining the ease-of-use, familiarity, availability across diverse platforms, and cost benefits, of Linux. Our overall approach exploits the popular batched-execution methodology currently employed for clusters of high-end computer systems. Using this model, application developers implement, debug, and test their codes using a development (non-production) machine and they submit the resulting application to a batch processing system that manages a cluster of production compute engines. When a sufficient portion of the cluster is available, the batch system allocates the number of nodes required for the application, installs and initializes the application (and its data) on the cluster resources, and invokes the program with the specified parameters.

Using a batch execution model, applications run *in isolation* on the cluster. Thus, if the cost of installation is not too great, it should be possible to install a customized Linux kernel for each application when it is launched. For long-running scientific applications, this start-up overhead should be negligible. In the same light, it may be that machines are dedicated to the execution of one single application or suite of applications. We wish to explore the performance impact of building (automatically eventually) Linux kernel instances that are specifically tuned for a small

collection of applications.

In this work, we first profile the application using a particular parameterization and data set to identify system calls that are frequently executed. We then modify the Linux kernel by-hand to specialize these calls according to how they are used by the program. Our focus in this paper is on specializing the Linux I/O subsystem. Our specializations bypass much of the general-purpose code that Linux employs along the critical path of the I/O system call sequence. In addition, we investigate the impact of reducing the Linux image so that it contains only the necessary functionality required by the application.

We focus on disk file write behavior. While much research has centered on improving read performance (c.f. Section 5 for a survey of the related work) we notice that many scientific codes are affected by write performance due to the need to checkpoint periodically. In the following sections, we describe the applications that we consider as well as the analyses that we use to characterize their I/O file write behavior. We then show how we exploit this behavior to reduce the overhead of the I/O subsystem in Linux for our application's specific IO pattern.

3 Application Specific Behavior

In this section, we describe our behavior analysis methodology, the two applications that we consider, and the results of our analysis for each. The applications are a small I/O benchmark that we developed and the MIT implementation of the General Circulation Model (MITGCM) [**?**].

3.1 Behavior Analysis Methodology

To specialize the Linux operating system for an application, we need to understand its full system dynamic behavior. Therefore, we extract two dynamic system analyses from each application: an annotated, full system (application and operating system), dynamic call graph and an analysis of the I/O patterns. Both analyses are program-independent and do not require modification to the source program or the availability of the program source.

To generate the former, we employ KernProf [**?**], an open-source full-system profiler from Silicon Graphics Incorporated. KernProf consists of a kernel patch that deploys a number of profile data collection mechanisms and a device driver that controls them. It implements a user-level commands that allow users to configure and control the kernel profiling functionality. KernProf introduces some overhead for profile collection, however this overhead is distributed evenly over each dynamic function invocation. We feed the KernProf output to GNU gprof [**?**] which converts the profile information into the gprof format. Gprof is an application-level profiler that is part of the Linux GNU binutils library. We produce the annotated kernel call graph from the gprof output, using a tool that we have developed, and use the DOT [**?**] utility to visualize the call graph. The annotations are counts for each edge that the program traverses during execution and the total time spent in each function.

We note that other kernel profiling capabilities are available, but none that are as facile as KernProf. The *Oprofile* utility [**?**] that is available for most modern Linux kernels can capture system behavior, but cannot instrument specific callpaths when specific applications are running. We believe that the KernInst [**?**] utility would have provided a second option but we found that we could not, ultimately, make it function properly in our environment. The choice of KernProf, however, limits our analysis to Linux 2.4 and 2.5 kernels exclusively and the 2.95 version of gcc. We discuss some of the effects of these limitations in Subsection 4.1.

The second analysis that we perform is the extraction of I/O patterns. In particular, we identify patterns in the way an application exercises the Linux I/O system during execution. Our analysis produces patterns from the number of unique files read and written, the size of the files, the num-

User Time	System Time	Real Time
5.5645sec	38.117sec	44.3sec

Table 1. Execution time breakdown for the simple benchmark code. The times are the average over 20 executions of the benchmark on a dedicated x86-based 2.8GHz machine running Linux 2.4.18.

ber of bytes read and written on each I/O system call, and the total number of system calls invoked. The analysis identifies the the most frequent patterns the the program uses to access the I/O subsystem. We employ both analyses to characterize the I/O behavior of an application and to identify opportunities for customization and specialization of the Linux system according to the application's requirements and behavior.

3.2 I/O Benchmark

The first application that we investigate is a simple program that writes an 80MB file to a local EXT2 Linux file system. The program executes a single loop that repeats 10,000,000 times. Each iteration of the loop executes a write call with an 8-byte string parameter. This benchmark simulates the write behavior of an application that writes a small amount of data to a file at a time, e.g., for logging or check-pointing during execution.

Table 1 shows the average execution time breakdown of this benchmark across 20 runs (using a dedicated, x86-based 2.8GHz machine). The kernel executes for 38 seconds (86% of the total time) on average during execution of the program. The most frequent system call made by the benchmark is to the sys_write function. The total time of the benchmark consists mainly of the time spent in the write call and the time spent crossing the user-kernel boundary.

Figure 1 shows a portion of the Linux 2.4.18 dynamic call graph from our I/O analysis for the benchmark program. Each oval represents a function and we annotate each oval with two values. "self" is the time the kernel spends executing the function on behalf of the user program, and "children" is time the kernel spends executing the function's children, i.e., the callees, of the function. We only show a subset of children in this figure. An arrow represents a call edge from a caller (arrow source) to a callee (arrow destination). The value on an edge shows the number of times the kernel traverses the edge during execution.

The kernel spends 34 seconds (88% of system time) executing sys_write and its children on behalf of the application. The kernel calls all the functions along this path approximately 10,000,000 times, calling them each time the application executes a write operation.

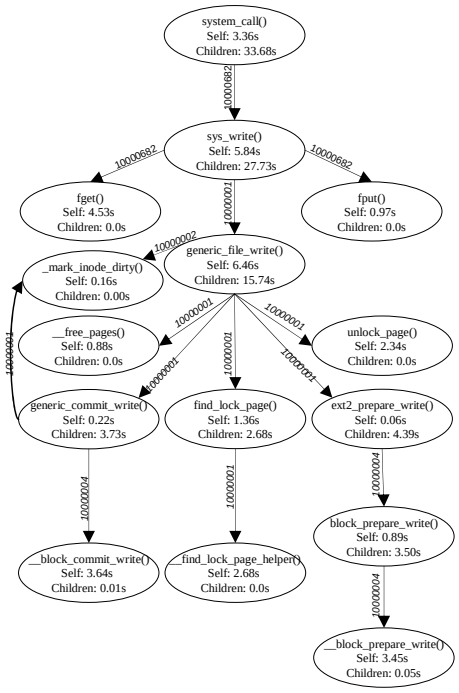

Figure 1. Subgraph from the full-system, dynamic call graph of the Linux 2.4.18 kernel executing on behalf of the benchmark program

There are two important functions along the sys_write call path: fget() and fput(). fget() is the kernel function that is responsible for converting

a process file descriptor to a file object and for returning the latter to the process. fput() is the kernel function that cleans up after a process is finished using the file object. When the file usage counter becomes 0, fput releases the the internal representation (resource) of the file object.

The call graph indicates that fget and fput are called every time sys_write is called. However, fget returns the same file object each time. In addition, fput is invoked needlessly for all but the last invocation of sys_write. fget and fput account for 6 seconds of the total execution time (14%) on average.

find_lock_page() and __find_lock_page_helper() are two other functions in which the kernel spends 11% (4 seconds) of its time on average. These functions find pages on which the new bytes should be written. This process requires traversal over all of the processes' pages; however, the page often returned is the same page as returned by the last call to the function sequence.

The I/O pattern analysis for the benchmark is simple since the program employs only a single write system call. The pattern is a large number of sequential writes, each very small in size. Moreover, the execution of this pattern repeats itself over the lifetime of the program.

This small example shows that there is much redundancy and unnecessary processing required for managing file writes in the Linux kernel despite the predictability (repetition) of the I/O pattern for this code. The kernel is designed in this way such to avoid saving process-specific state and to maintain the integrity and safety of the file system in a multi-program environment. However, when a single process accesses a particular file in some memory chunks, this policy constitutes unnecessarily performance overhead.

3.3 MITGCM Ocean Circulation

The second application that we consider is an implementation of the General Circulation Model (GCM), a popular numerical model used by application scientists to study oceanographic and climatologic phenomena. GCM simulates ocean and wind currents and their circulation in the earth atmosphere thousands of years in advance. A widely used implementation of GCM is made available by MIT Climate Modelling Initiative (CMI) team [**?**, **?**, **?**]. Researchers commonly integrate this implementation into oceanographic simulations, as in [**?**, **?**, **?**]. The MIT CMI team supports a publically available version that they package with several test inputs, which are carefully optimized to decrease any overhead and enhance system resources' utilization.

We employ the `exp2` for this study. `exp2` simulates the planetary ocean circulation at a 4 degree resolution. The simulation uses twenty layers on the vertical grid, ranging in thickness between 50m at the surface to 815m at depth. We configure the experiment to simulate 365 days of ocean circulation at a one-second resolution.

The I/O behavior of the simulation involves reading several input file at the beginning of the run for initialization, check-pointing processed data at equal time intervals in sequential order to several files, and outputting the final results to another file. The total number of write system calls that the application invokes is 179,579. The number of read system calls that the program executes is 630. Table 2 shows the average execution time breakdown across 40 runs of the MITGCM application. 95% of the total time is spent in user space for computation. 4% of the total is spent in system time; it is this portion of the execution time on which we focus.

Figure 2 shows a portion of the dynamic call graph for Linux 2.4.18 kernel when executing on behalf of MITGCM. sys_write() and sys_llseek() are the most frequently called system calls. They constitute 82% of time spent within system_call and 19% of total system time. On the other hand, the asynchronous call_do_IRQ() and its children consume 53% of the system time, which is not shown in this portion of the graph.

sys_write() and sys_llseek() both call fget() and

User Time	System Time	Real Time
170.887sec	7.71974sec	180.4sec

Table 2. Execution time breakdown for the MITGCM Ocean Circulation code. The times are the average over 40 executions of the benchmark on a dedicated x86-based 2.8GHz machine running Linux 2.4.18.

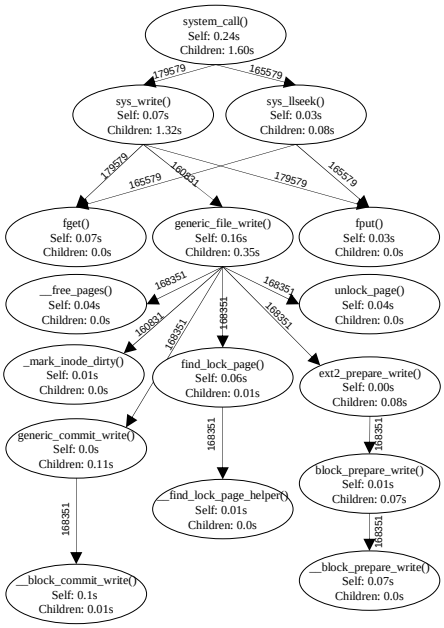

Figure 2. Subgraph from the full-system, dynamic call graph of the MITGCM application

fput() on every file access and return the same file object for every call sequence. Moreover, as in the benchmark program, the invocation of sys_write triggers invocation of each of its children, e.g. find_lock_page(), _mark_inode_dirty() and other functions along the sys_write call path. __block_commit_write() for MITGCM is also frequently invoked as part of this path. In this function, the kernel scans all pages involved in the write operation, in order to map the data from user space to kernel space. Moreover, the kernel performs a second scan to count the pages with their dirty bit set. This process introduces additional I/O overhead that is specific to the I/O pattern of MITGCM code, which degrades performance.

We categorize the I/O behavior of MITGCM using three patterns. For all patterns, MITGCM opens a single file at a time, reads from or writes to it, and closes the file before opening another file. The first we call `bin-files`. In this pattern, MITGCM consumes data input from several input files with extension `bin`. MITGCM reads each of these files, 4 kilobytes at a time, and repeats this process until execution terminates. The second pattern, which we refer to as `meta-files`, writes files of type `meta`. These files are small in size (183-212 bytes) and MITGCM writes all of the data to each file in this category as one chunk using one sys_write call. For the final pattern, which we call `data-files`, MITGCM writes to large files, the size of which can exceed 1 megabyte. MITGCM writes to these files at a rate of 180-360 bytes per call to sys_write. This pattern accounts for 98% of the total bytes that MITGCM writes and for almost 100% of the number of calls to the sys_write function.

Indeed, while we have presented the benchmark application before MITGCM in this paper for clarity, we in fact designed the benchmark application based on this analysis. Specifically, the benchmark we constructed to make a series of small, short disk writes to mimic only the short-write behavior of the MITGCM code. In this way,

	bin-files pattern	meta-files pattern	data-files pattern	All patterns
Total bytes read/written	648000	102940	32356800	33107740
% of bytes read/written	1.957%	0.310%	97.731%	100%
Num. of calls to sys_write	0	508	160240	160240
% of sys_write calls	0%	0.316%	99.683%	100%
Number of files	7	508	508	1023

Table 3. MITGCM I/O patterns

we can examine the effect of specialization on the specific pattern of interest in isolation, and also *in situ*.

Table 3 summarizes the MITGCM I/O pattern data. These analyses expose opportunities for us to exploit application-specific behavior to customize the Linux kernel and reduce I/O overhead. We next describe how we use these analyses to do so.

4 Specialization And Results

We investigate two different forms of specialization for the MITGCM and benchmark codes. In the first, we use the behavioral characteristics that we extract using our analyses to avoid performance bottlenecks of the programs. To enable this, we use the analyses to *customize* the Linux kernel according to the specific usage patterns of the applications. In particular, we focus on the file write patterns of each application and introduce buffering to avoid the system call overhead as we described earlier. It is clear from the data in Table 1 that buffering writes should improve the performance of the benchmark. For the MITGCM code which has been optimized to avoid I/O as a bottleneck, the expected improvement should be less.

We also evaluate the performance of these codes when we reduce the footprint of the Linux kernel by eliminating unused components from the system. In particular, we model the case where each code is to be executed on a dedicated machine sequentially. Thus we disable all subsystems except the serial interface (needed for the console) and the disk system. This form of specialization, we believe, is most appropriate for CPU-intensive programs such as the MITGCM code where memory and cache performance are critical. By eliminating unused kernel subsystems, this specialization should reduce page-table and swapping overhead as well as TLB pressure.

Table 4 shows the in core memory footprint (in megabytes) for all of the configurations we test in this work. The original kernel, which we refer to in the table and elsewhere in this paper, uses the default configuration provided by the menuconfig tool, including the modules support. The Modified kernel is the Linux kernel with the buffered short disk writes, as it is described in the next subsection. The third column describes the slim-kernel size, a kernel configured with the basic support for only the necessary devices and subsystems for the correct operation of the applications we are considering here. Slim-kernel would not be useful to run other applications since it is missing support for many basic devices. The fourth column illustrates the memory size of the kernel image for a slim-kernel with the special-

Kernel size in Mbytes	Original kernel	Modified kernel	Slim kernel	Slim+Modified kernel
2.4.18 kernel footprint	1.86	1.86	0.57	0.567
2.6.10 kernel footprint	4.182	4.182	1.423	1.423

Table 4. Kernel memory footprint sizes (in megabytes) for 2.4.18 and 2.6.10 kernels.

ized I/O system call.

Also, we test two different kernel versions: 2.4.18 and 2.6.10. To generate the call-graph information described in Section 3 we needed to use 2.4.18 as the *kernprof* utility is only available for that kernel version. We realize that most kernels in use today, however, track the latest stable releases and, thus, we also example 2.6.10. However, since we cannot extract the same hot-path information for 2.6.10, we are relying on the analysis of 2.4.18 to be applicable to the later kernel. As we describe below, this assumption is most likely invalid. To keep the comparison as controlled as possible, however, we use gcc 2.95 with the 2.6.10 kernel in all experiments even though a more modern version of gcc is available.

4.1 Buffering Short Disk Writes

Our analysis of the benchmark application indicates that a significant source of overhead on the program is imposed by the repeated invocation of the write system call. The overhead results from the user-kernel boundary crossing and the redundant operations that we identify in the previous section.

To avoid this overhead, we investigate two implementations of buffering. In the first, we modify the program itself to perform the buffering. This specialization amounts to an application-level-only optimization and should represent an upper-bound on what can be achieved by a kernel-level specialization designed for the same access pattern. Instead of performing the individual write calls, the benchmark program buffers the data in memory. The program then writes the buffer to disk when the buffer fills.

The second implementation of buffering that we investigate, is to modify the kernel to perform buffering while at the same time maintaining the correct Linux I/O call semantics. Our customization allocates a new buffer of size 4096 bytes in kernel memory when the application opens the file. We modify the write call so that when the application performs subsequent writes to the file which are less than 4096 bytes in length, the routine stores them in the buffer and returns from the system call instead of invoking the full Linux system write apparatus. Only when the buffer is full, the application closes the file, or the application flushes the file does the routine send the data to disk through the regular file-system code. Thus, this process eliminates the calling sequence of the children of the sys_write system call in the dynamic call graph for every short write waiting, instead, until there is a full page worth of data to be written. We also added an additional file position pointer to the file struct. The new file position pointer tracks the point in the file to which the buffer should be written. Thus this pointer ensures that we maintain the integrity of the file data. We selected 4096 since it is the virtual page size in our system hoping that it would be treated efficiently by the virtual memory subsystem.

For each method, we record the average wall-clock time reported by the Unix time utility and we compute the average improvement over the "stock" Linux 2.4.18 kernel over 20 runs. Unsurprisingly, the results show substantial improve-

ment in both cases. For the simple benchmark, the application-level buffering reduces system time, on average by 97%, i.e., buffering avoids both the overhead of user-kernel boundary crossing and of performing redundant work in the call sequence of file writes within Linux itself. Kernel-level buffering reduces system time by 66% on average. Kernel-level buffering is unable to reduce overhead to the same degree as application-level buffering since the program must still cross the user-kernel boundary repeatedly on each sys_write call. This boundary crossing accounts for a significant portion of the overhead. This result motivates and provides insight into the potential of customization techniques for which we can move part of the application execution *into* the kernel itself to avoid the boundary crossing overhead.

4.2 Performance Comparison Among Kernel Approaches

Having determined that a kernel-level intermediate buffering scheme can generate appreciable performance gains, we detail the wall-clock performance of the benchmark and the MITGCM codes using two different versions of the Linux kernel: 2.4.18 and 2.6.10. In both cases we use the Fedora Core FC1 Linux distribution and gcc 2.95.3 on a 2.8 GHz Pentium IV with 40 gigabytes of disk attached, a 100 megabit ethernet connection and 512 Mbytes of RAM.

Figure 3 shows the relative performance of four different execution configurations for the simple I/O benchmark using each kernel. The left-most verticle bar (labelled *original kernel*) in each case, shows the base-line performance of an unmodified kernel installation with a default configuration. The bar second from the left (labelled *Modified kernel*) shows the performance of the benchmark when the kernel is customized to include a buffer for short disk writes. The bar third from the left indicates the performance of the benchmark when the kernel has been "slimmed" (marked *slim kernel* in the figure) to include only those features that are needed to execute the program. Finally, the full specialized version (marked *Slim+Modified kernel* in the figure) indicates the performance when the kernel is configured with the buffering customization and as a slim kernel. Furthermore, we use t-test on our data to measure the statistical significance of our averages.

On the Linux 2.4.18 kernel, the buffering customization improves execution performance of the simple I/O benchmark from 44.3 seconds to 17.4 seconds as expected. What is somewhat surprising is that the slim kernel without the buffering (11.8 seconds) customization out performs the customized but unslimmed version (17.4 seconds). Each number represents an average over 20 separate runs after system initialization. We computed a t statistic of 4.14 over the data which indicates that the difference in the observed average performance between these two cases is statistically significant at the 0.05 significance level. By the same token, the average observed performance of the slimmed kernel and the slimmed kernel with the buffering customization (12.8 seconds in the figure) results in a t statistic of 0.68 indicating that the results are not statistically distinguishable at the 0.05 significance level.

For the 2.6.10 kernel, however, the results are quite different. In this case, the buffer customization generates the most significant improvements. Using the short-write buffer improves performance from 19.7 seconds to 10.17 seconds whereas the slimmed kernel without the buffer customization generates an average run time of 14.8 seconds. These differences are all statistically significant at the 0.05 significance level when considered pairwise. However, the difference between the unslimmed, customized kernel (10.17 seconds) and the Slim+Modified kernel (10.3 seconds in the figure) fails a hypothesis test at the 0.05 significance level. Thus, unlike the case for the 2.4.18 kernel, for 2.6.10, the buffer customization (and not kernel slimming) is

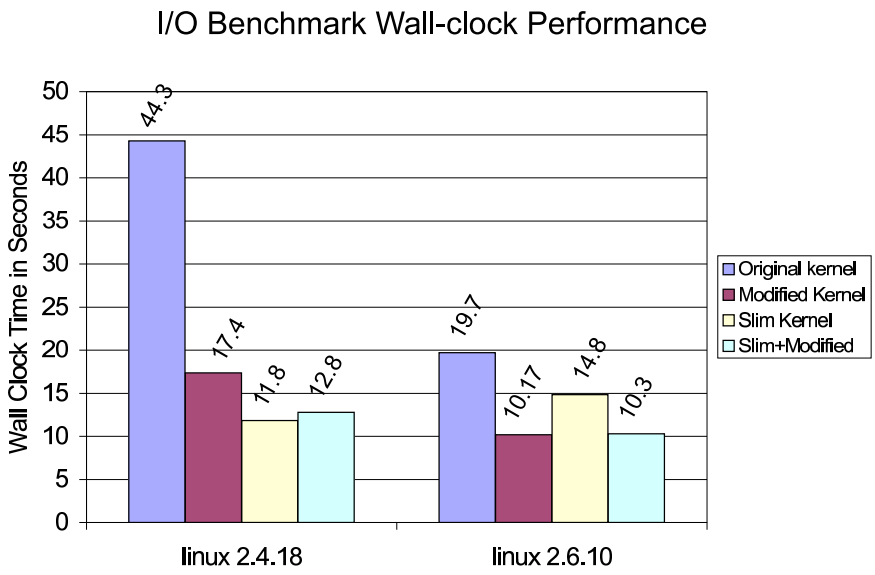

Figure 3. Comparison of wall-clock timings for I/O benchmark using buffering customization and kernel slimming for Linux 2.4.18 and Linux 2.6.10 kernels.

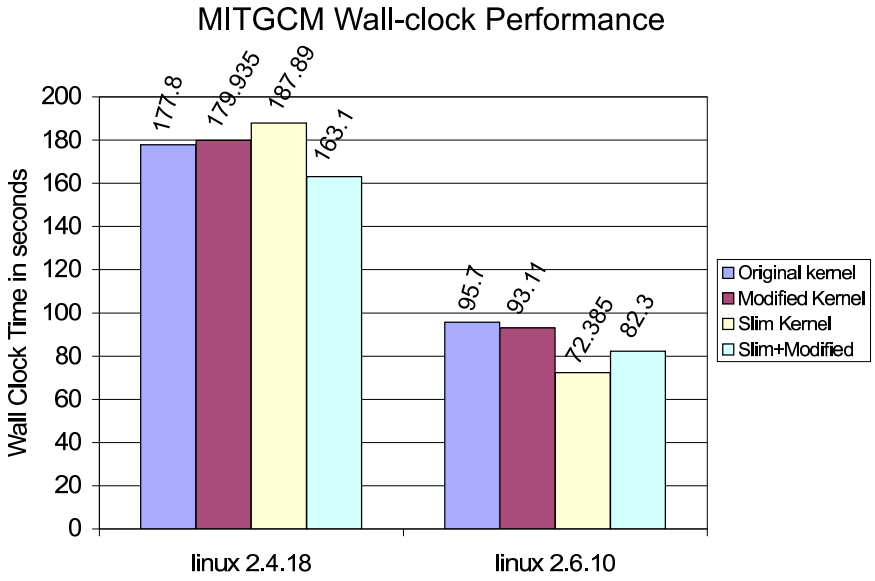

Figure 4. Comparison of wall-clock timings for MITGCM benchmark using buffering customization and kernel slimming for Linux 2.4.18 and Linux 2.6.10 kernels.

the most performance impacting optimization.

Figure 4 shows the same comparison for the MITGCM test code. For the 2.4.18 kernel, the average Slim+Modified times are lowest, but the only statistically detectable difference at the 0.05 significance level is between the largest time (in this case 187.89 seconds for the slimmed kernel) and the smallest time (163.1 seconds for the Slim+Modified kernel). In the 2.6.10 case, the uncustomized slimmed kernel performs best, and the differences between all averages except the original kernel and the customized but unslimmed kernel are statistically significant at the 0.05 significance level.

4.3 Discussion

From this data, it is difficult to identify a single customization strategy that works for both test codes on both kernels. Even though we developed the simple disk-write benchmark based on our observation of I/O behavior in the MITGCM code, the best customization strategy in each case appears to be different. We can conclude, however, that the performance of both of these codes can be significantly affected by the choice of customization method. In all cases, there is at least one detectable difference in average wall-clock performance between the methods we investigate. Moreover, in the cases where the differences are significant, they are often a large portion of the total execution time. For example, the average execution time for the MITGCM code on the 2.6.10 kernel is reduced from 95.7 seconds to 72.385 seconds – a savings of approximately 24%. For weather modeling and prediction codes, such as MITGCM, which tend to be long-running, CPU-intensive codes, it is somewhat surprising that such a large gain in performance can be achieved through a reduction in the footprint size of the kernel.

To test the theory that the slimmed kernel is using the cache and/or TLB resources more efficiently, we recompiled the MITGCM code using the PAPI [**?**] profiling tool so that we could interrogate the hardware performance counters supporting by the CPU. Figure 5 compares the cache-miss and TLB-miss counts in all configurations on the 2.6.10 kernel. Note that PAPI requires gcc 3.2.3 or greater. Thus the accuracy of these results may be affected since the original experiments were conducted using gcc 2.95.

From the figure, it appears that both cache and TLB performance is improved as a result of kernel slimming. Additionally, we suspected that the short-buffer modification would have a negative effect on cache-performance since it introduces an additional copy of each datum. This affect is visible in the L2 cache miss column, but is not as noticeable for the L1 cache. We do notice a slight improvement in TLB miss rate with the slimmed versions, but overall it is difficult to believe that the 24% improvement shown in Figure 4 results from improved memory system performance. We are exploring different hypotheses for how the minimalist configurations are able to achieve large performance gains, but in the context of this work, these results lead us to pursue profile-based approaches. It is clear that some of the subsystems interact in a performance retarding way, even when those subsystems are not used directly by the running applications. Profiling seems a more promising methodology for capturing these interactions than static analysis alone.

5 Related Research

Operating system (OS) customization and specialization techniques have been proposed, studied, and evaluated extensively over the past decade and a half [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. Two primary approaches to these OSs are microkernels [**?**, **?**, **?**, **?**, **?**, **?**] and systems that allow user code to be injected into the kernel [**?**, **?**, **?**, **?**, **?**, **?**].

These systems have gained limited acceptance and use due to the performance overheads they introduce and the significant change in programming model and environment that they require of

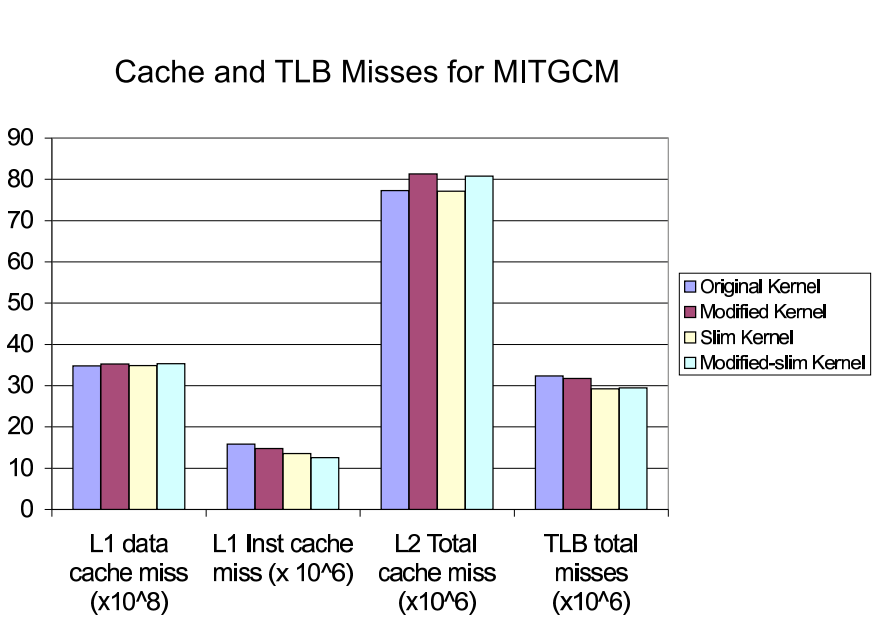

Figure 5. Cache-miss and TLB-miss counts for MITGCM as Measured by PAPI.

users. These systems take the approach that extensibility is the common case. As a result, these systems introduce significant overheads for interprocess communication and memory sharing, expensive system calls, dynamic linking of new services, and context switching overhead. Moreover, these systems require applications to be written for the specific OS implementation at hand (requiring a new programming methodology) and to *select* the versions of services that will provide them the most performance benefit. That is, these systems require that applications be expertly programmed to use the OS functionality correctly and efficiently and that the application programmer be able to integrate the application with the OS explicitly. Our approach maintains the current programming model without modification (thus supporting legacy codes and enabling programmer productivity).

An alternative approach that is more similar to our own is to customize an existing operating system for common application behaviors [**?**, **?**, **?**, **?**]. These systems perform specialization using partial evaluation of system call parameters to reduce the length of critical paths through the kernel. Such systems systems focus on *specializing* existing OS code and automatically infer when specialized version should be employed. In addition, as different applications execute, specialized code is either selected using a template mechanism [**?**, **?**, **?**, **?**, **?**] or *dynamically replaced* with new versions [**?**, **?**, **?**, **?**, **?**, **?**].

Our work differs from this prior research in that we focus our efforts on scientific applications and the familiar and ubiquitous Linux operating system. Prior work has either developed entirely new systems or have extended upon proprietary OSs (AIX [**?**], HP-UX [**?**], and Solaris [**?**]). Moreover, these systems focus on enabling general-purpose, multi-program execution. We assume an isolated application model – one that is commonly used for batched execution of scientific codes using cluster resources, which significantly simplify the OS customization process. We need not be concerned with cross-application interference, resource sharing, and conflicting customizations. As a result, we can focus on aggressive, profile-guided, customization and footprint reduction of Linux that is application-specific and low overhead.

Reducing the size of the Linux footprint is the focus of prior work primarily in the area of

resource-constrained systems. For example, the researchers in [**?**] and [**?**], show that reducing the size of Linux is effective for running Linux on embedded systems. Chanet et al [**?**] employs link-time binary rewriting for compacting and specializing the kernel for embedded systems. This specialization is different from ours since they do not modify the system call policy and instead use partial evaluation of system call parameters and unneeded system calls elimination. The researchers in this prior work show that they are able to reduce Linux version 2.4.19 from 1095KB to 925KB; however, their results show that doing so introduces 4% performance overhead on average. We show that we can concurrently reduce the Linux footprint and improve execution performance. To our knowledge, we are the first researchers to consider footprint reduction for OSs intended for high-end application execution.

Finally, other prior work has identified opportunities for I/O customization by showing empirically that many scientific workloads and application behavior is not well supported by the current I/O policies of modern OSs, including Linux [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. The work described in [**?**] is similar to our customization of short file writes. However, they modify the application to perform specialization (as we describe initially for the I/O benchmark in Section 4.1). We show that kernel modification performs similarly to application modification, however the former does not require the use or understanding of the source code. Moreover, our work employs hot, full-system, dynamic call paths to identify opportunities for customization and implements Linux specialization that is guided by this profile information and I/O pattern analysis.

6 Conclusions and Future Work

We believe that this work demonstrates the potential performance benefits of kernel specialization. Even for CPU-intensive programs such as the MITGCM code, substantial performance gains are possible if the application is running in isolation, and the kernel has been specialized specifically for it.

At the same time, it is clear that determining what specializations will prove most profitable is a difficult problem. Even when detailed information about application behavior in the kernel is available (through intensive call-path analysis) specializations designed to improve performance in one application may retard performance in another that exhibits similar behavior. Moreover, the performance benefits are sensitive to the version of the kernel employed and, perhaps more alarmingly, the specific kernel configurations that are chosen. Given the combinatoric complexity associated with testing all possible kernel configurations for some version of the Linux kernel, we believe that a more profile-guided analysis technique is needed. Thus one future direction we will explore focuses on the analysis techniques needed to specify application-specific kernel specializations.

We also plan to explore how specialized kernels can be employed in batch systems. Using high-speed Linux configurators (such as SDSC Rocks [**?**]). We believe it is possible to automate kernel installation with an acceptable level of overhead. Doing so while maintaining the integrity of the overall system is also a focus of our current and future work.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the Performance of I/O-Intensive Parallel Applications. In *Workshop on I/O in Parallel and Distributed Systems*, 1996.

[2] A. Adcroft, J. Campin, P. Heimbach, C. Hill, and J. Marshall. *MIT-GCM User Manual*. Earth, Atmospheric and Planetary Sciences, Massachusetts Institute of Technology, 2002.

[3] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Schedular Activations: Effective Kernel Support for User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[4] A. Banerji and D. Cohn. An Infrastructure for Application-Specific Customization. In *ACM European SIGOPS Workshop*, Sept. 1994.

[5] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance of the SPIN Operating System. In *Symposium on Operating System Principles*, Dec. 1995.

[6] D. Black, G. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel Operating System Architecture and Mach. In *Workshop on Micro-Kernels and Other Kernel Architectures*, Apr. 1992.

[7] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of SuperComputing 2000 (SC'00)*, Nov. 2000.

[8] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the linux kernel. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES0̀5)*, pages 95–104, Chicago, Jun 2005.

[9] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E. Volanschi, J. Lawall, and J. Noye. Tempo: Specializing Systems Applications and Beyond. *ACM Computing Surveys*, 30(3), 1998.

[10] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys*, 34(4):450–468, 2002.

[11] D. Engler, M. Kaashoek, and J. O. Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating System Principles*, Dec. 1995.

[12] S. Graham, P. Kessler, and M. McKusick. Gprof: An execution profiler for modular programs. *Software Practice & Experience 13*, pages 671–685, 1983.

[13] Graphviz: Graph visualization software. `http://www.graphviz.org`.

[14] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A flexible base of distributed programming. In *Symposium on Operating System Principles*, Dec. 1993.

[15] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[16] C. Herbaut, J. Sirven, and S. Fevrier. Response of a simplified oceanic general circulation model to idealized NAO-like stochastic forcing. *Journal of Physical Oceanography*, 32, 2002.

[17] D. Hildebrand. QNX: Microkernel Technology for Open Systems Handheld Computing. In *Proceedings of Pen and Portable Computing Conference Exposition*, May 1994. `http://www.qnx.com/`.

[18] Kernprof: SGI Kernel Profiller. `http://oss.sgi.com/projects/kernprof/`.

[19] C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee. An application-oriented linux kernel customization for embedded systems. *J. Inf. Sci. Eng.*, 20(6):1093–1107, 2004.

[20] J. Levon. OProfile - a system profiler for linux, 2004. `http://oprofile.sourceforge.net/`.

[21] Y. Li, S. Tan, M. Sefika, R. Campbell, and W. Liao. Dynamic Customization in the Choices Operating System. In *Reflection Conference*, Apr. 1996.

[22] J. Liedke. On Micro-Kernel Construction. In *Symposium on Operating System Principles*, Dec. 1995.

[23] M. Mackall. Linux-tiny And Directions for Small Systems. In *Proceedings of the Linux symposium*, July 2004.

[24] T. Madhyastha. *Automatic Classification of Input/Output Access Patterns*. PhD thesis, University of Illinois, Urbana-Champaign, August 1997.

[25] R. Marlet, C. Counsel, and P. Boinot. Efficient Incremental Run-Time Specialization for Free. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[26] J. Marotzke and R. G. et al. Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport sensitivity. *Journal of Geophysical Research*, 104(C12), 1999.

[27] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. In *Symposium on Operating System Principles*, Dec. 1989.

[28] MIT's Climate Modeling Initiative. `http://paoc.mit.edu/cmi/`.

[29] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, and T. Proebsting. Scout: A Communications-Oriented Operating System. In *Hot OS*, May 1995.

[30] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba – A distributed Operating System for the 1990's. *IEEE Computer*, 23(5), May 1990.

[31] K. Nisancioglu, M. Raymo, and P. Stone. Reorganization of Miocene deep water circulation in response to the shoaling of the Central American Seaway. *Paleoceanography journal*, 18, 2003.

[32] F. Noel, L. Hornof, C. Counsel, and J. Lawall. Automatic, Template-Based Run-Time Specialization: Implementation and Experimental Study. In *International Conference on Computer Languages*, 1998.

[33] NPACI Rocks Cluster Distribution. `http://www.rocksclusters.org/Rocks`.

[34] B. Pasquale and G. Polyzos. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proceedings of Supercomputing conference*, Nov. 1993.

[35] B. Pasquale and G. Polyzos. A case study of scientific application I/O behavior. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Jan. 1994.

[36] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Mateo, CA, 1994.

[37] C. Pu, T. Autrey, A. Black, C. Counsel, C. Cowan, J. Inouya, L. Kethana, J. Wapole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating System Principles*, Dec. 1995.

[38] C. Pu, H. Massalin, and J. Ioannidis. The Synthetics Kernel. *Computing Systems*, 3(1), 1990.

[39] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A System Software Kernel. In *Computer Society International Conferance*, Feb. 1989.

[40] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guilemont, F. Herrman, C. Kaiser, S. Langois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Apr. 1992.

[41] C. Small and M. Seltzer. VINO: An integrated platform for operating system and database research. Technical Report Technical Report TR-30-94, Harvard Univ. Cambridge, MA, 1994.

[42] C. Small and M. Seltzer. Self-monitoring and Self-adapting Operating Systems. In *IEEE Workshop on Hot Topics in Operating Systems*, May 1997.

[43] D. Stammer, C. Wunsch, R. Giering, C. Eckert, P. Heimbach, J. Marotzke, A. Adcroft, C. Hill, and J. Marshall. Global ocean circulation during 1992-1997, estimated from ocean observations and a general circulation model. *Journal of Geophysical Research-Oceans*, 107, 2002.

[44] A. Tamches and B. Miller. Using Dynamic Kernel Instrumentation for Kernel and Application Tuning. *International Journal of High-Performance and Applications*, 13(3), 1999.

[45] S. Tan, D. Raila, and R. Campbell. An Object-Oriented Nano-Kernel for Operating System Hardware Support. In *International Workshop on Object Orientation in Operating Systems (IWOOOS)*, Aug. 1995.

[46] Y. Wang and D. Kaeli. Profile-guided I/O Partitioning. In *Supercomputing*, Nov. 2003.

[47] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1992.