Cassyopia: Compiler Assisted System Optimization

Mohan Rajagopalan Saumya K. Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{*mohan, debray*}@*cs.arizona.edu*

Matti A. Hiltunen Richard D. Schlichting
AT&T Labs-Research
180 Park Avenue
Florham Park, NJ 07932, USA
{*hiltunen, rick*}@*research.att.com*

Abstract

Execution of a program almost always involves multiple address spaces, possibly across separate machines. Here, an approach to reducing such costs using compiler optimization techniques is presented. This paper elaborates on the overall vision, and as a concrete example, describes how this compiler assisted approach can be applied to the optimization of system call performance on a single host. Preliminary results suggest that this approach has the potential to improve performance significantly depending on the program's system call behavior.

1 Introduction

Execution of a program almost always involves multiple address spaces, whether it executes on a standalone system such as a desktop or a PDA or across multiple machines such as a client-server program or a program based on the Web or the Grid. On a standalone system, user-level address spaces (processes) request services from the kernel address space using system calls and potentially interact with other user-space processes or system daemons. Programs that span machines by definition are composed of multiple processes, which also interact with kernels, e.g., to exchange messages. Address spaces play a valuable role as protection boundaries, and typically represent units of independent development, compilation, and linking. Largely for these reasons, crossing address spaces—even on the same machine—has considerable execution cost, typically orders of magnitude higher than the cost of a procedure call [9, 10].

Here, we describe an approach to reducing this cost—sometimes dramatically—using compiler optimization techniques. Unlike traditional uses of such techniques that are confined to optimizing within procedures (intra-procedural optimization), across procedures (inter-procedural optimization), or across compilation units (whole-program optimization), our approach focuses on applying these techniques across address spaces on the same or different machines while preserving the desirable features of separate address spaces. The specific focus is on *profiling-based optimization* where the address space crossing behavior of a component (e.g., the system calls made by a process) is profiled and then optimized by reducing the number of crossings or the cost of each crossing. This compiler assisted approach to system optimization is being realized in a system called Cassyopia.

This paper elaborates on this overall vision. We first highlight its application in one context, that of optimizing traditional system call performance on a single host. This work complements existing techniques for system call optimization [3, 4, 9, 11, 12, 13], which focus on optimizing calls in isolation rather than collections of multiple calls as done here. We then briefly discuss other areas in which these ideas could be applied.

2 Case Study: System Call Clustering

Overview. As an application of the general approach described above, we describe *system call clustering*, a profile-directed approach to optimizing a program's system call behavior. In this approach, execution profiles are used to identify groups of systems calls that can be replaced by a single call implementing their combined functionality, thereby reducing the number of kernel boundary crossings. A key aspect of the approach is that the optimized system calls need not be consecutive statements in the program or even within the same procedure. Rather, we exploit correctness preserving compiler transformations such as code motion, function inlining, and loop unrolling to maximize the number and size of the clusters that can be optimized. The single combined system call is then constructed using a new *multi-call* mechanism that is implemented using kernel extension facilities like those described in [1, 2, 5, 6, 12, 17]. We also introduce *looped multi-calls*, an extension to the basic technique spanning address spaces. The overall approach is illustrated by a simple copy program.

Clustering mechanisms. This section describes the mechanisms used to realize system call clustering in a traditionally structured operating system. The goal, in addition to reducing boundary crossing costs, is to reduce the number of boundary crossings required. Specifically, we want to extend the kernel to allow the execution of a sequence of system calls in a single boundary crossing. The new mechanism must not compromise protection, transparency or portability, significant advantages provided by the existing system call mechanism. We now look at one such mechanism, the *multi-call* [14].

A multi-call is a mechanism that allows multiple system calls to be performed on a single kernel crossing, thereby reducing the overall execution overhead. Multi-calls can be implemented as a kernel level stub that executes a sequence of system calls. At the application level, the multi-call interface resembles a standard system call and uses the same mechanism to perform the kernel boundary crossing, thereby retaining the desirable features of the system call abstraction. An ordered list of system calls to be executed is passed as a parameter to the multi-call. Each system call in the list is described by its system call number and parameters. An issue that has to be addressed to preserve correctness is that of error behavior, i.e., replacing a group of system calls by a multi-call must not alter the original error behavior of the program. Upon detecting an error in any constituent system call, the multi-call returns control to the application level and reports the system call in which the error occurred as well as the error itself.

Modifications to a program to replace a sequence of system calls by a multi-call are conceptually simple and can be done using a compiler. In the next section, we briefly discuss profiling and the use of compilation techniques to add multi-calls to the program.

Profiling. Given this mechanism, the issue becomes one of identifying optimization opportunities in the program, both in the sense of identifying sequences of calls that can be replaced by a multi-call and identifying correctness-preserving program transformations that can be used to create such sequences. Profiling does this by characterizing the dynamic system call behavior of a program on a given set of inputs. Operating system kernels often have utilities for generating such traces (e.g., `strace` in Linux), or they can be obtained by instrumenting kernel entry points to write to a log file.

Figure 1.c shows a graphical summary of a collection of such traces for a simple file copy program; the code for this program is in figure 1.a, while 1.b shows the control flow. Each node in the system call graph represents a system call with a given set of arguments. Consecutive system calls in the trace appear as nodes connected by directed edges indicating the order. The weight of each edge indicates the number of times the sequence appears. This graph forms the basis for compile-time transformations for grouping system calls. The general idea is to find frequently executed sequences of calls in the system call graph; if the corresponding system calls are not syntactically adjacent in the program source, we attempt to restructure the program code so as to make them adjacent, as described below.

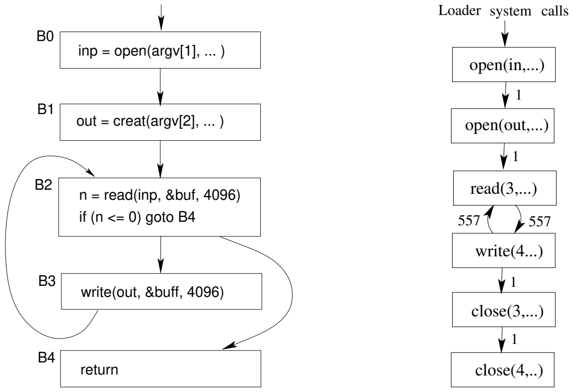
Applying compiler optimizations. The fact that two system calls appear as a sequence in the graph does not, by itself, imply that the system calls can be grouped together. This is because even if two system calls follow each other in the trace, the system calls in the program code may be separated by arbitrary user code that does not include system calls. Replacing these calls by a multi-call would require moving the intervening code into the multi-call, which may compromise safety. To increase the applicability of this technique, we use simple, well-understood, correctness preserving transformations like function inlining, code motion and loop unrolling that enhance the applicability of our optimization. Although code rearrangement is a common compiler transformation, to our knowledge it has not been used to optimize system calls as done here. Here, we give examples of program transformations that can be used to rearrange the statements in a program to allow system call grouping without affecting the observable behavior of the program.

A simple such transformation involves interchanging *independent statements*. Two statements are said to be independent if neither one reads from or writes to any variable that may be written to by the other. Two adjacent statements that are independent and have no externally visible side-effects may be interchanged without affecting a program's observable behavior. This transformation can be used to move two system calls in a program closer to each other, so as to allow them to be grouped into a multi-call. Note that such system calls may actually start out residing in different procedures, but can be brought together (and hence, optimized) using techniques such as function inlining.

Another useful transformation is *loop unrolling*. In the control flow graph (figure 1.b), the `if` statement in basic block B4 prevents the `read` and the `write` system calls from being grouped together. Programs like FTP, encryption programs, and compression programs (e.g., gzip and pzip) exhibit similar control dependencies. In cases like this where the dependency appears within a loop, loop unrolling can sometimes be used to eliminate the dependency. In the case of the copy program in fig-

```
#include <stdio.h>
#include <fcntl.h>

#define N 4096

void main(int argc, char* argv[])
{
  int inp, out, n;
  char buff[N];

  inp = open(argv[1],O_RDONLY);
  out = creat(argv[2],0666);

  while ((n = read(inp,&buff,N)) > 0) {
    write(out,&buff,n);
  }
}
```

(a) Source code



(b) Control flow graph (c) Syscall graph

Figure 1: Copy program

	Entry	Exit
System Call	140 (173-33)	189 (222-33)
Procedure Call	3 (36-33)	4 (37-33)

Table 1: CPU cycles for entry and exit

ure 1, for example, unrolling the loop once and combining the footer of one iteration with the header of the next iteration results in the code shown below, with adjacent system calls within the loop that are now candidates for the multi-call optimization:

```
n = read(in, buff0, size);
while (n > 0) {
    write(outfd, buff1, n);
    n = read(in, buff0, size);
}
```

Looped multi-calls. The *looped multi-call* is a variant of the basic multi-call motivated by this philosophy. It is applicable in the situation where, after other transformations have been applied, the entire body of a loop consists of a single multi-call. In this case, the entire loop is, in effect, moved into the kernel by replacing it by a looped multi-call. This results in a single kernel boundary crossing rather than one per iteration. For example, in the copy program, notice that after replacing the write-read sequence, the body of the loop contains just one multi-call. The loop can now be moved into the kernel by using the looped multi-call. Notice again that the semantics of the program are not affected by this transformation.

Experimental results. A number of experiments have been performed to identify both the potential and actual benefits of this approach. All tests were run on a Pentium II-266 Mhz laptop running Linux 2.4.4-2.

As a baseline, we first measured the cost of a system call versus a procedure call. Table 1 gives the results of these experiments; these results were obtained using the `rdtscl` call, which reads the lower half of the 64 bit hardware counter Read Time Stamp Counter, RDTSC provided on Intel Pentium processors. These results indicate that clustering even two system calls and replacing them with a multi-call can result in savings of over 300 cycles every time the pair of system calls is executed.

Table 2 gives the results of applying system call clustering using both the multi-call and the looped multi-call to the copy program shown in figure 1. To do this, the multi-call or looped multi-call was assigned system call number 240 and added as a loadable kernel module. The numbers reported in Figure 2 were calculated by taking the average of 10 runs on files of 3 sizes ranging from a small 80K file to large files with size around 2Mb. The block was chosen as 4096 bytes since it was the page size and hence the optimal block size for both the optimized and unoptimized versions of the copy program. Smaller block sizes would result in a larger number of system calls and hence the multi-call would show larger relative improvement. The maximum benefit seemed to appear in the small and medium files since disk and memory operations seemed to dominate for larger files.

3 Other Compiler Assisted Techniques

The multi-call mechanism can be extended further to include code other than the system calls, error checking, and loops in the multi-call. Specifically, we can extend

File Size	Original	Multi-call		Looped Multi-Call	
	Cycles (10^6)	Cycles (10^6)	% Savings	Cycles (10^6)	% Savings
80K	0.3400	0.3264	4%	0.3185	6.3%
925K	4.371	4.235	3.1%	4.028	7.8%
2.28M	10.93	10.65	2.6%	10.37	5.2%

Table 2: Optimization of a copy program with block size of 4096.

the basic code-motion transformations to identify a *clusterable region*, possibly containing arbitrary code, that can then be added to the body of a multi-call. Optimization techniques like dead-code elimination, loop invariant elimination, redundancy elimination, and constant propagation can then be applied to optimize the program. For example, the data transformation code in programs such as compression or multimedia encoding/decoding can be included in the multi-call.

Another avenue of optimization is to replace general purpose code in the kernel by compiler-generated case-specific code in user-space. Examples of such general code are the register saves and restores executed by the kernel before and after each system call. Since the kernel does not know which registers are actually used by the application process, it must save and restore all of them. This can be quite expensive on processors with a large number of registers. However, the compiler has this information, and it can therefore generate specialized user-space code for saving and restoring registers. Simulations using this strategy for a 3 parameter `read` system call on the Intel StrongARM processor show up to 20% reduction in the number of cycles required to enter the kernel. Other such examples include the general permission checking performed by each system call. Note that both of the above extensions require that a degree of trust be placed in the compiler.

The profiling and compiler-based optimization can also be used to enable controlled information sharing between address spaces. Traditionally, components in different address spaces optimize their internal behavior not knowing what type of interactions will be received from other address spaces. For example, the operating system conserves battery power by switching hardware devices such as the CPU, display, hard disk, and wireless cards into power-saving modes based on a period of inactivity. These policies are generally based on statistical models of application behavior that attempt to predict future (in)activity based on patterns of past activity. Because of their stochastic nature, they can be quite inaccurate for individual applications, and result in significant performance overheads [18, 19, 20]. However, by carefully exposing some of the components internal state to other address spaces using a *translucent boundary API*, each address space can optimize its behavior to better match the requirements of other system components, and hence aim for a global optimum. The profiling and compiler techniques can be used to collect and generate the information at the application's translucent boundary API to the kernel. The same approach can be used for compiler assisted scheduling, where an adaptive scheduler can fine-tune the scheduling policy based on the processes running in the system and their requirements. The compiler could place "yield" points within the body of the program to indicate schedulable regions and changes in requirements. Conversely, if the kernel exposes changes in the state of an existing resource, e.g., a reduction in CPU speed to conserve power, the application process may be able to adapt its internal algorithms [7] to degrade the service gracefully while still satisfying user requirements.

Finally, note that the same principles can be applied to any address space crossing, including distributed programs where the "boundary crossing" cost involving network communication may have a delay of tens or hundreds of milliseconds. In particular, clustering multiple remote procedure calls (or remote method invocations in distributed object systems such as CORBA and Java RMI) can lead to significant savings. Furthermore, more general code movement techniques such as moving client code to the server or server code to the client when appropriate can also be used [21]. Note that the object migration techniques used in systems such as Emerald [8] have the same goal, but without the systematic support provided by our profiling and compiler techniques.

4 Concluding Remarks

The optimizations suggested in this paper complement existing system specialization techniques and may be used in conjunction with these techniques to yield even greater improvement. We are in the process of integrating these optimizations into the PLTO binary rewriting tool [16] as an optimization pass. Applying these techniques to the popular media-player `mpeg_play` [15] has yielded an average 25% improvement in the frame rate, 20% reduction in the execution time and 15% in CPU cycles, suggesting the potential of this approach.

We have identified several optimization targets ranging from utility programs like gzip and pzip, to web servers and database applications. Independent of the savings in CPU cycles, we believe such techniques will yield significant energy savings, greater in fact than what the reduction in CPU cycles would imply.

References

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

[2] R. Campbell and S. Tan. μ-Choices: An object-oriented multimedia operating system. In *Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, WA*, May 1995.

[3] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, Dec 1993.

[4] A. Edwards, G. Watson, J. Lumley, D. Banks, and C. Dalton. User space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *SIGCOMM*, Aug 1994.

[5] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference, Monterey, CA, USA*, June 1999.

[6] B Henderson. Linux loadable kernel module, HOWTO. http://www.tldp.org/HOWTO/Module-HOWTO/, Aug 2001.

[7] M. Hiltunen and R. Schlichting. A model for adaptive fault-tolerant systems. In K. Echtle, D. Hammer, and D. Powell, editors, *Proceedings of the 1st European Dependable Computing Conference (Lecture Notes in Computer Science 852)*, pages 3–20, Berlin, Germany, Oct 1994.

[8] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb 1988.

[9] J. Mauro and R. McDougall. *Solaris Internals-Core Kernel Architecture*, pages Section 2.4.2 (Fast Trap System Calls), 46–47. Sun Microsystems Press, Prentice Hall, 2001.

[10] J. Mogul and A. Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, 1991.

[11] C. Poellabauer, K. Schwan, and R. West. Lightweight kernel/user communication for real-time and multimedia applications. In *NOSSDAV'01*, Jun 2001.

[12] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–324, Copper Mountain, CO, Dec 1995.

[13] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[14] M. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. System call clustering: A profile-directed optimization technique. Technical report, The University of Arizona, 2002.

[15] L. Rowe, K. Patel, B. Smith, S. Smoot, and E. Hung. Mpeg video software decoder, 1996. http://bmrc.berkeley.edu/mpeg/mpegplay.html.

[16] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO : A link time optimizer for the Intel IA32 architecture. In *Proceedings of Workshop on Binary Rewriting*, Sept 2001.

[17] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Operating Systems Design and Implementation*, pages 213–227, 1996.

[18] T. Simunic, L. Benini, and G. DeMicheli. Event-driven power management of portable systems. *IEEE Transactions on Computer Aided Design*, July 2001.

[19] T. Simunic, L. Benini, P. Glynn, and G. DeMicheli. Dynamic power management of laptop hard disk. *DATE*, 2000.

[20] T. Simunic, H. Vikalo, P. Glynn, and G. DeMicheli. Energy efficient design of portable wireless systems. *ISPLED*, 2000.

[21] D. Waugaman and R. Schlichting. Using code shipping to optimize remote procedure call. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 17–24, Las Vegas, NV, Jul 1998.