Reducing the Energy Cost of Application/OS Interactions

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

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

Abstract

Software approaches to power optimization have traditionally had two distinct foci, in relative isolation, targeting either individual applications (compilation-based techniques) or global (operating system) policies. Dynamic interactions between the application and operating system through system calls, which can potentially have a large impact on overall performance and power consumption, remain largely unoptimized due to the partitioning of concerns. This paper discusses the energy implications of a new *system call clustering* optimization technique for reducing application/OS interaction costs that is based on a novel *multi-call* mechanism. Preliminary results on common utility programs such as the *mpeg_play* video decoder have been promising.

1 Introduction

The amount of energy consumed during program execution is becoming an increasingly important concern for a wide spectrum of computing systems, influencing issues ranging from battery lifetime to the amount of heat generated. Research on software approaches to power optimization have generally had two distinct foci: compiler optimizations to reduce the energy usage of an application [4, 7, 8, 14]; and operating system design to improve energy efficiency [3, 5, 9, 10, 16]. By and large, each of these efforts has been carried out in relative isolation, i.e., with little interaction between application-level compiler optimizations and operating system level optimizations. This partitioning of concerns causes missed opportunities for energy optimization. In this paper, we focus on reducing the overheads associated with the interaction between an application and the underlying operating system.

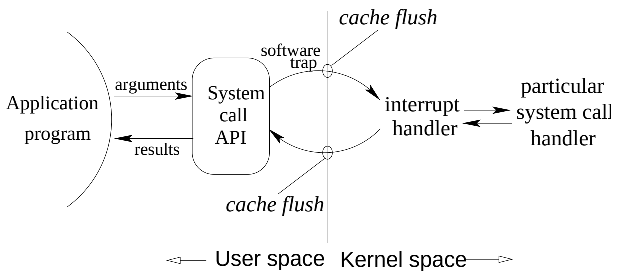


Figure 1: Schematic of a system call

A system call is significantly more expensive than an ordinary procedure call—more than 20 times the cost of regular procedure calls by one measure [11]. They are also heavily used in many types of programs, e.g., Web and FTP servers, media players, and utilities like *copy*, *gzip*, and *compress* use system calls to access files and sockets. This combination of cost and ubiquity means that optimization of system calls—both individually and for a program as a whole—can potentially have a large impact on overall program performance. Such optimizations are especially important for mobile devices such as cell phones, which have traditionally hosted task specific embedded applications but are now begining to support well defined general-purpose operating systems and applications.[1]

The operation of a system call is shown in figure 1. Much of the cost of making a system call comes from having to switch between distinct address spaces, from the user address space across the kernel boundary to the kernel space, and back. This

[1]For example, Motorola recently announced plans to have its A760 phone use Linux [2]; Texas Instruments and NEC have also backed the use of Linux in cell phones [1].

incurs a number of costs. First, there are the direct costs associated with the system call itself: those of passing the arguments to the call; checking these arguments for validity within the kernel; an indirect function call through an array of function pointers to access the handler for the call within the kernel; and communicating the return value back to the user application. Just as important are the indirect costs associated with switching the processor into kernel mode (when entering the kernel) and back (when leaving it), each of which flushes the cache. These operations have a high energy cost: both directly, because of the cost of the memory and indirect branch operations needed to enter the kernel and access the handler for the system call; and indirectly, because the cache flush and resulting cache misses on subsequent memory accesses consume a great deal of energy [15]. This suggests that the energy requirements of system-call-intensive programs can be reduced significantly if we can reduce the number of kernel boundary crossings.

This paper describes the application of a novel mechanism, called *multi-calls*, to reducing the energy requirements of system-call-intensive applications by reducing the number of kernel boundary crossings associated with a sequence of system calls. Our goal is to optimize both the operating system and applications executing on it to take advantage of knowledge of the nature of the dynamic interactions between the two, and thereby reduce the energy cost associated with crossing the application/OS boundary. We use execution profiles to identify frequently executed sequences of system calls, and use correctness-preserving transformations to replace such sequences, where possible, by a single call implementing their combined functionality, thereby reducing the number of kernel boundary crossings. The single combined system call is then constructed using a multi-call that is implemented using kernel extension facilities such as loadable kernel modules in Linux.

The remainder of this paper is organized as follows. Section 2 describes how the multi-call mechanism is used and briefly sketches how this mechanism is implemented in a conventional operating system. Section 3 gives some experimental results, and section 4 concludes.

2 Multi-Calls and System Call Clustering

A multi-call is a mechanism that allows multiple system calls to be performed on a single kernel crossing, thereby reducing the overall energy requirements of the program without compromising any of the advantages provided by the existing system call mechanism, namely, protection, transparency, and portability. 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. A list of system calls to be executed in specified order 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 in order to preserve correctness is that of error behavior—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.

Given this mechanism, the issues we have to address are as follows.

(i) Given a particular application to be optimized, how can we identify which system call sequences in that application should be candidates for optimization via a multi-call?

(ii) What should we then do to the application program to allow the use of multi-calls for these candidate sequences where possible?

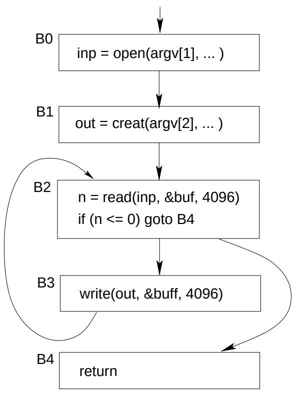
The first of these issues is handled via profiling, and is discussed in section 2.1. The second is done using semantics-preserving code transformations in a compiler (or similar program manipulation tool), and is discussed in section 2.2. Together, we refer to these techniques as *system call clustering*.

2.1 Profiling

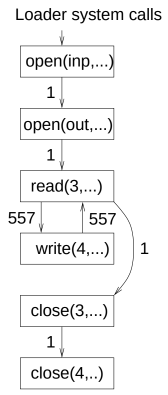
System call profiling is used to characterize the dynamic system call behavior of a program on a given set of inputs and thereby identify frequently occurring sequences of system calls that are candidates for multi-call optimization. Operating system kernels often have utilities for generating such traces (e.g., `strace` in Linux), or they can be obtained by instrumenting kernel entry points to write to a log file. The resulting system call trace is then analyzed to

```
#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 = create(argv[2],0666);

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

(a) Source code


B0
inp = open(argv[1], ...)
B1
out = creat(argv[2], ...)
B2
n = read(inp, &buf, 4096)
if (n <= 0) goto B4
B3
write(out, &buff, 4096)
B4
return

(b) Control flow graph



(c) Syscall graph

Figure 3: Copy program

```
SysCallGraph = $\emptyset$;
prev_syscall = syscallTrace→firstsyscall;
while not (end of syscallTrace) {
    syscall = syscallTrace→nextsyscall;
    if (prev_syscall,syscall) $\notin$ SysCallGraph {
        SysCallGraph += (prev_syscall,syscall);
        SysCallGraph(prev_syscall,syscall)→weight = 1;
    } else
        SysCallGraph(prev_syscall,syscall)→weight++;
    prev_syscall = syscall;
}
```

Figure 2: *GraphBuilder* algorithm.

identify frequently occurring system call sequences. We perform this analysis by constructing a *syscall graph* that indicates how frequently some system call s_i is immediately followed by some other system call s_j in the system call trace. Each system call along with select arguments is represented as a unique node in the graph, and a weighted directed edge $a \xrightarrow{w} b$ from node a to node b with weight w indicates that the profile trace has w occurrences where the system call a is followed immediately by the system call b. The algorithm for graph creation is described in figure 2. The algorithm simply traverses the trace and adds new edges (and the corresponding nodes, if necessary) or increases the weight of an existing edge, as appropriate.

The idea is illustrated in figure 3. Figure 3(a) shows the source code for a simple file copy program, figure 3(b) its control flow graph, and figure 3(c) the syscall graph. This graph forms the basis for compile-time transformations for grouping system calls, as described below.

Once we have obtained the syscall graph, we use a greedy algorithm, similar to one commonly used by compilers for profile-directed code layout [12], to identify hot paths in the syscall graph. The system calls on these hot paths are then candidates for optimization using multi-calls.

2.2 Code transformations

The fact that two system calls are contiguous in the syscall graph does not, *ipso facto*, imply that they can be implemented using a multi-call. 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 other user code. Replacing these calls by a multi-call would require moving the intervening user code into the multi-call as well. This would result in arbitrary user code executing within the kernel, potentially compromising 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.

Given a hot path σ in the syscall graph, our goal is to try and transform the program so that the sequence of system calls that are adjacent in σ are also adjacent, in the same order, in the program code. Once this has been accomplished, it is straightforward to replace this system call sequence by an appropriate multi-call. To this end, we proceed as follows.

1. We use repeated applications of function inlining, if necessary, to bring all the system calls in σ into the same function.

2. If there is any user code separating a pair of the system calls of interest, we use correctness-preserving program transformations to restructure the code and bring the system calls together.

   If the hot path σ contains an edge (a, b) and we are unable to restructure the program so as to remove all user code separating the system calls a and b and bring them together, then the edge (a, b) is deleted, thereby breaking σ into two smaller paths that are then processed as before.

The program transformations in step (2) above are driven by the goal of moving user code away from between a pair of system calls that are candidates for optimization. The following is a (non-exhaustive) set of examples illustrating how this is done.

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. Given code of the form

```
system_call_1
UserCode
system_call_2
```

If *UserCode* and *system_call_2* are independent and *UserCode* has no externally visible side-effects, we can restructure the code to obtain

```
system_call_1
system_call_2
UserCode
```

and similarly for *system_call_1* and *UserCode.*

Loop unrolling. Loop unrolling is used when the system calls of interest are in a loop, and there is user code between the two system calls, but where the loop body is (or can be rearranged to be) such that there is no user code before the first system call and after the last system call in the loop. The file copy program shown in figure 3, as well as applications such as FTP, encryption programs, compression (*gzip* and *pzip*) exhibit similar characteristics. In the case of the copy program of figure 3, 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(inp, &buff, N);
while (n > 0) {
    write(out, &buff, n);
    n = read(inp, &buff, N);
}
```

2.3 Applying system call clustering

Once a sequence of candidate system calls have been brought together in the program, they are replaced by a single multi-call with two arguments: the number of system calls being passed to it, and an array of structures where each element of the array describes one system call; each such entry consists of the system call number, arguments, a field for the return value, and a bit indicating whether that system call should be checked for an error.

2.4 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 the 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.

In theory, one can extend the basic code-motion transformations to identify a *clusterable region*, which 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 also potentially be used to maximize optimization opportunities.

Program		*No. of iterations*	% BATTERY CHARGE DRAINED		Improvement (%)
			Unoptimized (U)	*Optimized* (O)	$(U - O)/U$
mpeg_play	`dg1`	40	45	32	28.9
	`dg2`	40	44	33	25.0
	`dg5`	25	30	23	23.3
file_copy		40,000	67	60	10.4

Figure 4: Experimental results

2.5 Implementing multi-calls

Linux provides support for Loadable Kernel Modules that allow code to be added to the kernel without recompilation [6]. We use this functionality to add the new customized system calls needed by our approach. Such "new" calls are given system call numbers greater than 240. The use of loadable modules is comparable to compiling new system calls into the kernel as far as performance is concerned.

The value returned by the multi-call indicates whether any of the system calls produced an error. If an error is found to have occurred, the value returned by the multi-call identifies the source of the error. This can then be used to pinpoint the type of error encountered, after which the original error-handling code is invoked.

3 Experimental Results

This section describes the experiments and setup used to evaluate the potential and actual benefits of the approach. The testbed comprised of Pentium II-266 Mhz laptops with Li-ION batteries running Linux 2.4.4-2 with 96 Mb of RAM and APM BIOS 1.2 Kernel Driver 1.14. For each experiment, the battery was charged fully, and the program under consideration was run repeatedly; after each iteration the *apm* utility was used to check the battery status. Energy consumption due to other sources were minimized by using the default system power management—for example, the monitor would sleep after about a minute of inactivity.

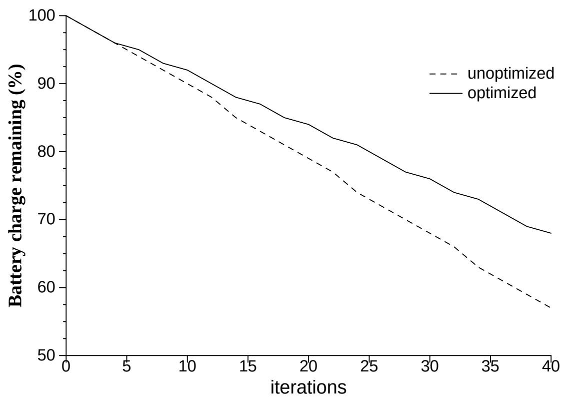
Figure 4 presents the experimental evaluation for two programs : *mpeg_play*, a popular mpeg video player, *file copy*, and a simple file copy program as described earlier (we will have additional benchmark results by the time the final version of the paper is due). Figure 5 illustrates the battery drain rates measured in our experiments: the y-axis represents the battery charge remaining (as a % of original charge) while the x-axis denotes the number of iterations. Figure 5(a) is for a mpeg video player, while Figure 5(b) is a plot for the copy program.

mpeg_play. The popular *mpeg_play* application [13], represents multi-media applications running on mobile devices. Three different files - dg1(2752 frames), dg2(2959 frames) and dg3 (3616 frames) were used as input. During the experiment, each file was run in a loop for several iterations. As described in the table the use of multi-calls improved battery consumption by an order of 25%.

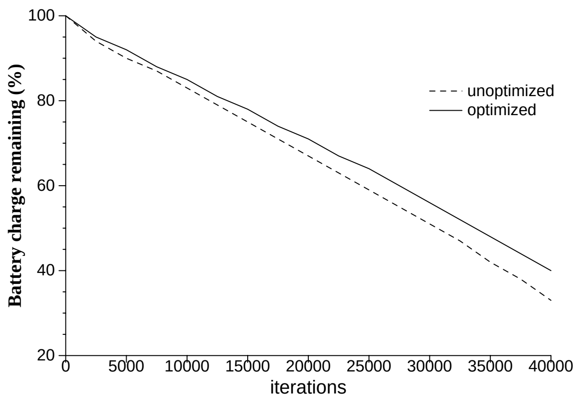
File copy. The copy program is representative of a class of utility programs such as compress, encrypt etc which exhibit a similar "read-followed-by-write" structure. Successive read and write calls were replaced by a single *read_write* multicall. The input file to the copy program was a 187K binary file. The program was repeated 40,000 times.

4 Conclusions

Traditionally, software approaches to reducing the energy usage of systems have focused either on applications or on the underlying operating system, with little attention paid to the interaction between applications and operating systems. It turns out, however, that this interaction point can be quite expensive in terms of energy consumption, because of the cost of switching from the user address space to the kernel address space and back. This paper addresses this issue by describing how the number of kernel crossings of an application can be reduced using system call clustering and a mechanism called the multi-call. The energy savings resulting from this optimization can be quite substantial in system-call-intensive programs: our experimental results indicates improvements in the amount of battery charge consumed ranging from over 10% for a file copy program, to almost 29% for an MPEG player.



(a) mpeg_play (input: `dg2`)



(b) file copy

Figure 5: Battery drain rate

References

[1] Linux coming to cell phones. c|net News.com, Jan. 10, 2003. `http://news.com.com/2100-1033-980214.html`.

[2] Motorola moving cell phones to linux. *eWeek*, Feb. 13 2003. `http://www.eweek.com/article2/0,3959,887377,00.asp`.

[3] A. Acquaviva, L. Benini, and B. Ricco. Energy characterization of embedded real-time operating systems. In *Proc. Workshop on Compilers and Operating Systems for Low Power 2001 (COLP'01)*, 2001.

[4] G. Araujo. *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, Princeton University, 1997.

[5] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Power analysis of embedded operating systems. In *Design Automation Conference*, pages 312–315, 2000.

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

[7] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proc. 37th Conference on Design Automation (DAC-00)*, pages 304–307, June 5–9 2000.

[8] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *Proc. International Symposium on System Synthesis*, pages 55–61, Oct 2000.

[9] S.-F. Li, R. Sutton, and J. Rabaey. Low power operating system for heterogeneous wireless communication systems. In *Proc. Workshop on Compilers and Operating Systems for Low Power 2001 (COLP'01)*, 2001.

[10] Y.-H. Lu, L. Benini, and G. De Michelli. Power-aware operating systems for interactive systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(2), April 2002.

[11] J. Mauro and R. McDougall. *Solaris Internals-Core Kernel Architecture*. Sun Microsystems Press, Prentice Hall, 2001.

[12] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

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

[14] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proc. IEEE 1994 Symposium on Low-Power Electronics*, 1994.

[15] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, December 1994.

[16] A. Vahdat, A. Lebeck, and C. Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. In *Proc. SIGOPS European Workshop*, Sep 2000.