System Call Clustering: A Profile-Directed Optimization Technique

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

Techniques for optimizing system calls are potentially significant given the typically high overhead of the mechanism and the number of invocations found in many programs. Here, a profile-directed approach to optimizing a program's system call behavior called *system call clustering* is presented. In this approach, profiles are used to identify groups of systems calls that can be replaced by a single call, thereby reducing the number of kernel boundary crossings. The number and size of clusters that can be optimized is maximized by exploiting correctness preserving compiler transformations such as code motion, function inlining, and loop unrolling. This paper describes the algorithmic basics of system call clustering and presents initial experimental results performed on Linux using a new mechanism called *multi-calls*. The sample programs include a simple file copy program and the well-known mpeg_play video software decoder. Applying the approach to the latter program yielded an average 25% improvement in frame rate, 20% reduction in execution time, and 15% reduction in the number of cycles, suggesting the potential of this technique.

1 Introduction

Minimizing the overhead of system calls is one of the most fundamental goals in the design and implementation of operating systems. Not only are system calls expensive—more than 20 times the cost of regular procedure calls by one measure [11]—they are also widely used. 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.

This paper describes *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 such as loadable kernel modules in Linux.

System call clustering is useful for many types of programs, especially those that exhibit repetitive system call behavior such as Web and FTP servers, media players, and utilities like copy, gzip, and compress. Moreover, this technique can be exploited in different ways depending on the ability or desire to customize the kernel. At one level, once the basic multi-call mechanism has been installed in a kernel, it can be used directly by programmers to optimize sequences of system calls in a straightforward way. However, multi-calls themselves can also be customized, specialized, and optimized, essentially resulting in the ability to automatically extract collections of systems calls and associated well-defined pieces of code and insert them in the kernel. This could be used, for example, to highly optimize the performance of a device dedicated to a given application or small set of applications, such as a mobile video device. Note also that the approach has value beyond simple performance improvement, including as a technique for optimizing power usage in battery-powered devices such as laptops and PDAs.

The primary goals of this paper are to describe the algorithmic basics of system call clustering and to present initial experimental results that suggest the potential of the approach. While the technique generalizes across a wide variety of operating system platforms, the concrete focus here is on describing its realization for Linux and its application to sample programs that include a simple file copy program and the well-known mpeg_play video software decoder [21]. As an example of the value of the approach, applying system call clustering to the latter program resulted in an average 25% improvement in frame rate, 20% reduction in execution time, and 15% reduction in the number of cycles. We also highlight a number of other attributes of our solution, including simplicity and the ability to be easily automated. This work complements existing techniques for system call optimization, which tend to focus on optimizing the performance of calls in isolation rather than across multiple calls as done here [12, 16, 18, 19].

The remainder of the paper is organized as follows. Section 2 provides background on systems calls and introduces multi-calls as the basic mechanism for implementing system call clustering. This is followed in section 3 by a description of our basic approach, including the profiling scheme, clustering strategies, and compiler optimization techniques. Section 4 gives experimental results that demonstrate the improvement that can result from application of our approach. This is followed by discussion and related work in section 5. Finally, section 6 offers conclusions.

2 Clustering Mechanisms

This section describes the mechanisms used to realize system call clustering, and in particular, the multi-call mechanism that allows multiple calls to be replaced by a single call in a traditionally structured operating system such as Linux. It also provides background on system calls and their specifics in Linux. For the sake of concreteness, the discussion here considers the implementation of Linux on the Intel Pentium architecture.

2.1 Background

A system call provides a mechanism for crossing the user/kernel protection boundary in a controlled manner. The steps used to do this are typically as follows. First, the parameters required by the system call are stored on the stack or in pre-defined registers. These parameters include the user-level parameters, as well as the identity (number) of the system call to be invoked. Next, the processor is switched into kernel mode using a software interrupt or a trap instruction. The interrupt handler in the kernel then simply locates the correct system-call handler—a procedure stored in kernel space—based on the system call number and invokes this handler. The system-call handler typically first checks the call parameters for validity and then executes its task. Any results are propagated back to the caller the same way as the parameters (i.e., through the stack or registers). If an error occurs, an error number describing the error is passed back to the caller.

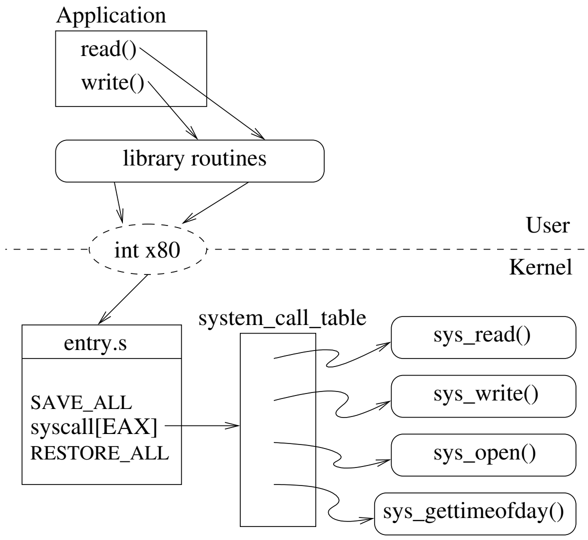


Figure 1: System calls in Linux

System calls in Linux are similar to the above. The mapping between system call names and numbers is provided simply by defining for each system call syscall_name a unique constant value of the form of __NR_syscall_name (in file asm/unistd.h). Within the kernel, all handlers are procedures that are named as sys_syscall_name by convention. These handlers can be accessed through an array of function pointers system_call_table that is indexed by the system call number. The actual number of system calls varies between Linux versions, but the Redhat Linux 2.4.2-2 version used here includes 221 predefined calls.

Figure 1 illustrates the sequence of events that occurs during a system call on Linux. The application program simply makes a function call to the _libc library with the appropriate parameters. The library call is responsible for marshaling arguments, loading registers, and issuing the trap instruction (int x80) that transfers control to the kernel. Parameters are passed either by value or by reference. By convention, register EAX is used to store the system call number. All the kernel entry procedures for Intel x86 architectures are stored in a file named arch/arch-type/kernel/entry.S. The entry procedure for system calls involves saving the state of the caller using the macro SAVE_ALL, followed by a call to the system call handler through system_call_table[EAX](). Upon completion, the caller's context is restored and results returned using the RESTORE_ALL macro.

Linux also provides support for Loadable Kernel Modules that allow code to be added to the kernel without recompilation [9]. We use this functionality to add the new customized system calls needed by our clustering approach. Note that the use of loadable modules is comparable to compiling new system calls into the kernel as far as performance is concerned.

2.2 Multi-Calls

While system calls provide significant advantages related to protection, transparency, and portability, they also incur considerable execution overhead resulting from the steps outlined above. For example, on Solaris the latency of system calls is 22 times greater than that of procedure calls [11]. Our experiments indicate that the system call overhead relative to procedure calls is similar on Linux (see section 4).

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 standard system calls, as shown in figure 2. The multi-call has two arguments, the number of basic system calls in this invocation (count) and an array of structures (sys_params), where each entry describes one system call. Each entry consists of the system call number, parameters, a field for the return value of the system call (result), and a field indicating if the system call should be checked for an error (i.e., return value < 0) (check_return_value). Both the parameters and the return value are passed by reference. get_params is a macro that retrieves the system call parameters from the parameter list.

The multi-call function iterates count times executing each system call in the order it is entered in the sys_params structure. If a system call returns an error, the multi-call either returns or continues execution depending on whether the specific system call is to be checked for errors or not. Note that not checking for errors corresponds to the case where the original program issuing the original system call did not check for errors after the call. The multi-call returns the number the first system call that failed (and was checked) or, in case no system call fails, count + 1. In our Linux experiments, a multi-call is implemented using a loadable kernel module and was assigned the unused system call number 240.

```
struct sys_params {
    int sys_call_no; // identity of the syscall
    void *params[]; // parameters of the syscall
    int *result; // return value of the syscall
    int check_return_value; // syscall be checked for errors?
}

int multi_call(int count, struct sys_params* args) {
    int i = 1; boolean error_occurred = false;
    while (i ≤ count) {
        sys_call = sys_call_table[args[i].sys_call_no];
        result = sys_call(get_params(args[i]));
        *(args[i].result) = result;
        if (result < 0 and args[i].check_return_value) {
            error_occurred = true;
            break;
        }
        i++;
    }
    if error_occurred return(i);
    else return(count + 1);
}
```

Figure 2: Multi-call Stub

Note that the basic multi-call mechanism can be extended to handle more complicated combinations of system calls, including cases where there are conditional branches with different system calls in each. The algorithms in section 3 precisely define the limitations on the type of functionality that can be included in multi-calls.

The modifications to a program to replace a simple sequence of system calls by a multi-call are conceptually straightforward. Figure 3 provides a simple example of an original code segment and one where a two system call sequence is replaced by a multi-call. The result of the multi-call indicates which, if any, original system call returned an error value, and thus it can be used to determine what error handling code is to be executed. The return value of the corresponding system call is returned in the result field of the parameter structure. Note that the details of the transformation depend on the specifics of the original program (see section 3). Our implementation uses a simple user-level wrapper function that takes care of marshaling arguments into the parameter structure, simplifying the modified code.

```
Original:
  res = write(out,buff,write_size);
  if res < 0 {
      error handling of write with error code res;
  } else {
      res = read(in,buff,read_size);
      if res < 0 {
          error handling of read with error code res;
      }
  }

Same program segment using multi-call:
  sys_params args[2]; int results[2];
  args[1].sys_call_no = __NR_write;
  args[1].params = [&out, &buff, &write_size];
  args[1].check_return_value = true;
  args[1].result = &results[1];
  args[2].sys_call_no = __NR_read;
  args[2].params = [&in, &buff, &read_size];
  args[2].check_return_value = true;
  args[2].result = &results[2];
  res = multi_call(2,args);
  if (res == 1) {
      error handling of write with error code results[1];
  } else if (res == 2) {
      error handling of read with error code results[2];
  }
```

Figure 3: Original and Optimized Program

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

Figure 4: *GraphBuilder* algorithm.

3 Profiling and Compilation Techniques

This section describes how profiling and compiler techniques can be used to optimize system call clustering using the multi-call mechanism. First, we describe the profiling technique that is used to identify frequently occurring system call sequences in a program. We then describe how to identify which portions of a program can be moved into a multi-call. This is followed by a discussion of compiler techniques that can be used to transform the program to enhance the applicability of our optimization by clustering system calls together, so that they can be replaced by a multi-call. Finally, we discuss the use of compiler techniques for specializing multi-calls and optimizing their execution.

3.1 Profiling

Profiling characterizes the dynamic system call behavior of a program. Operating system kernels typically have a single point of entry that can be instrumented to log an entry each time a call occurs to obtain the required profile. The Linux operating system provides a utility, strace, that provides this information. The output of strace includes the system call name, arguments and its return value. The system call trace of a program can be generated using the following command:

strace -e signal=none program *args*

The system call trace produced by strace provides the sequence of system calls executed in a run of the program, but this data must be further analyzed to identify frequently occurring system call sequences that are candidates for optimization. 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. Directed edges connecting nodes s_i and s_j are weighted by the number of times s_i is immediately followed by s_j in the trace. The frequently occurring system call sequences, the candidates for optimization, can then simply be identified based on edge weights.

The algorithm for graph creation is described in figure 4. 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. A detailed description of this technique is given in [20].

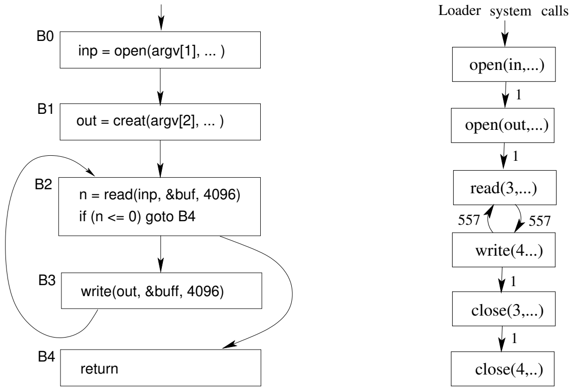
Figure 5 shows the source code for a simple file copy program, its control flow graph, and the syscall graph resulting from a typical execution of the program.

3.2 Identifying Multi-Call Code

This section address the question of how we determine which portions of the program can be implemented using multi-calls and which portions remain in user code. Intuitively, we want to identify a fragment of the program that can be pulled out into a function that can be executed within the kernel without compromising either the

```
#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 5: Example: a copy program

correctness of the program, or the security of the overall system. In order for a piece of code to be made into a function, it must have a well-defined entry point. In order to ensure that this code can be safely executed within the kernel, we have to make sure arbitrary user code cannot slip into a multi-call. We also have to guarantee that pulling out such code from an application program into a multi-call will not change the behavior of the program.

Based on this intuition, we define a *clusterable region* R in a program to be a portion of the program that satisfies the five conditions listed below. The first three are straightforward:

1. **Dominance.** There is a statement in R that dominates all statements in R.[1]

2. **Safety.** R contains only code that (i) sets up (e.g., evaluates arguments) and makes system calls; (ii) retrieves the return values of system calls; and (iii) makes control flow decisions based on such return values.

3. **Convexity.** Suppose the program contains statements A, B, and C such that A dominates B and B dominates C, then if A and C are in R then B must also be in R.

The first condition specifies that R has a well-defined entry point. The second condition precludes the possibility of arbitrary user code executing in the kernel. The third conditions ensures that abstracting R into a function that executes within the kernel will not unacceptably alter the order in which program statements executed.

The fourth condition involves a subtlety involving argument evaluation. Consider a pair of system calls

n = read(infp, &buf, 4096); write(outfp, &buf, n);

that we want to replace with a multi-call. In the original code, the arguments of each of these system calls are evaluated just prior to the call itself, so the value of the third argument in the call write(outfp, &buf, n) is that computed by the immediately preceding call to read. Now if we use a multi-call, and evaluate all of the arguments to these calls before making the multi-call, we will evaluate n before the execution of the read system call, and thereby obtain a potentially incorrect value. This suggests that the arguments of the system calls in the multi-call should somehow be evaluated after entering the multi-call, rather then beforehand. However, if system call arguments are allowed to be arbitrary expressions, then the only way to allow this is to either have an interpreter within the multi-call to evaluate arguments, or pass into the multi-call a code snippet for evaluating each argument; moreover, these expressions have to be evaluated in the caller's environment rather than the multi-call's environment. This means that to handle arbitrary expressions, we need a parameter-passing mechanism involving closures rather than simple values. Not only is this cumbersome and expensive, it also opens up a back door to allowing the execution of arbitrary user code within the kernel. For this reason, we disallow the use of arbitrary expressions as system call arguments in

[1] A node x *dominates* a node y in the control flow graph of a function if every path from the entry node of the function to y passes through x. Algorithms for computing dominators may be found in standard texts on compiler design (e.g., see [1]).

Notation: Given an edge $a \to b$ in a syscall graph, we use $Between(a, b)$ to denote the following collection of code: (i) the code to set up and make the system calls a and b; (ii) the code to retrieve the return values of these system calls; and (iii) any program statement x such that a dominates x and x dominates b in the control flow graph for the program.

Algorithm:

```
C = ∅;
for (each edge e ≡ 'a → b' in the syscall graph of the program, in descending order of weight){
    if (Between(a, b) violates any of the five conditions listed) continue;
    if (a is in an existing clusterable region C) {
        if (Between(a, b) can be added to C without violating the five conditions listed) {
            expand region C by adding Between(a, b) to it;
        }
    }
    else if (b is in an existing clusterable region C) {
        ... analogous to the previous case ...
    }
    else {      /* neither a nor b is part of an existing cluster */
        create a new region C consisting of just Between(a, b);
    }
}
```

Figure 6: An algorithm for identifying multi-call fragments

clusterable regions, limiting ourselves instead to a class of expressions that covers most situations commonly encountered in practice while permitting simple and efficient implementation. This is made explicit in our fourth condition:

4. Argument Safety. Each of the arguments of any system call in R is either a constant, a variable whose value is defined outside the region R, or a value computed by a previous system call in R.

A code region R satisfying these conditions can be abstracted into a function that can be executed within the kernel without compromising either the safety of the system or the correctness of the program. For reasons of simplicity, general applicability, and efficiency of implementation, however, the multi-call mechanism described in section 2.2 (see figure 2) accommodates only straight-line sequences of system calls. Our final condition on clusterable regions captures this:

5. Linearity. Control flow between the system calls in R is linear.

Relaxing this restriction is discussed in Section 3.4.

Figure 6 gives an algorithm for identifying code fragments in a program that can be converted to multi-calls.

3.3 System Call Clustering

The fact that two system calls are adjacent in the syscall graph does not, in itself, imply that they can be replaced by a multi-call. This is because even if two system calls follow one another in the syscall graph, the system calls in the program code may be separated by arbitrary other code that does not include system calls. If we replace these calls by a multi-call, we would have to move the intervening code into the multi-call as well, which would cause them to execute in the kernel, which may not be safe. To increase the applicability of multi-calls, we propose simple correctness-preserving transformations that enhance the applicability of our optimization. Although rearrangement of code is a common compiler transformation, to our knowledge it has not been used to optimize system calls.

Note that two adjacent system calls in the syscall graph may actually reside in different procedures. However, they can be brought together using simple techniques such as function inlining. Here, we assume function inlining has been performed if necessary and focus on more powerful transformations that actually change the order of statements in the program code.

3.3.1 Interchanging Independent Statements

A correctness preserving code motion transformation must ensure that dependencies between statements are not violated. Two statements S_1 and S_2 in a program are *independent* if (i) S_1 does not read from, or write to, any variable or object that is written to by S_2; and (ii) S_2 does not read from, or write to, any variable or object that is written to by S_1. Independence of statements can make it possible to rearrange code in a way that opens up avenues for system call clustering. Consider a code fragment of the form

syscall_1(); $stmt_1$; $\ldots$ $stmt_{k-1}$; $stmt_k$; syscall_2();

If $stmt_k$ and syscall_2() are independent, and $stmt_k$ has no side effects,[2] then they can be interchanged. The resulting code is of the form

syscall_1(); $stmt_1$; $\ldots$ $stmt_{k-1}$; syscall_2(); $stmt_k$;

This transformation can be used to bring system calls closer together, in effect moving out of the way user code that cannot be allowed to execute in the kernel. An analogous transformation can be defined to interchange syscall_1() and $stmt_1$ when they are independent and $stmt_1$ has no side effects.

The following code segment illustrates this transformation. This example includes three system calls: read, write, and sendto. In the simplest case, two system calls follow one another in the program code, statements 1 (read) and 2 (write). A more common case is when system calls are separated by a series of statements, in this example, sendto follows write directly in the syscall graph, but these system calls are separated by other statements (statements 3 and 4).

```
1. n = read(infd, buff, 4096);
2. write(outfd, buff, n);
3. read_bytes += n;
4. block_sent++;
5. sendto(sock_no, buff, n, 0, sock_addr, sock_len);
```

Statement 5 can be seen to be independent of statements 3 and 4, so it can be safely moved upwards to the point immediately after statement 2. This results in the following code segment, where statements 1, 2, and 5 may be replaced with a multi-call.

```
1. n = read(infd, buff, 4096);
2. write(outfd, buff, n);
5. sendto(sock_no, buff, n, 0, sock_addr, sock_len);
3. read_bytes += n;
4. block_sent++;
```

[2]Strictly speaking we also require that $stmt_k$ be guaranteed to terminate.

3.3.2 Restructuring Mutually Exclusive Statements

Our second transformation involves restructuring mutually exclusive code fragments. Consider code of the form

```
n = syscall_1();
if ( n < 0 )
    error_handler
syscall_2();
```

If *error_handler* can have side effects, we cannot interchange the **if** statement containing this code with syscall_2(). This is true even if the two statements are independent, since system calls typically have side effects, and if *error_handler* also has side effects, changing the order of side effects can potentially alter the behavior of the program. This is a problem, because we cannot move *error_handler*—which may contain arbitrary user code—into a multi-call; however, if we do not do so, then since the statement **if** $\ldots$ *error_handler* dominates syscall_2(), this code fragment will not satisfy the convexity criterion of section 3.2, and as a result this code will not be optimized to a multi-call.

We can get around this problem if the two statements are mutually exclusive, i.e., one will be executed if and only if the other is not (this may be, e.g., because *error_handler* eventually calls **exit**() or **abort**()). If mutual exclusion can be guaranteed, the program can be transformed to

```
n = syscall_1();
if ( n < 0 )
    error_handler
else
    syscall_2();
```

Bringing syscall_2() into the scope of the **if** statement, by incorporating it into its **else**-branch, changes the control structure of the program in such a way that *error_handler* no longer dominates syscall_2(). This can allow syscall_2() to be incorporated into a clusterable region together with syscall_1() and part of the **if** statement.

Identifying statements that are mutually exclusive is nontrivial. As a result, this transformation is more difficult to automate than that described in section 3.3.1. However, it may be implemented manually by a programmer who understands the high-level semantics of the code.

Comment: Given a tuple of arguments $\bar{a}$ to a system call, the function set_params($\bar{a}$) creates an array of pointers corresponding to the arguments $\bar{a}$, as follows. For each argument a_i in $\bar{a}$, if a_i is a constant c, set_params allocates a memory word w whose value is initialized to c and sets the i^{th} element of this array to point to w. If a_i is a variable or a value computed by a preceding system call, set_params this array element is set to point to the corresponding location.

Algorithm:

```
let K = the maximum number of system calls in any clusterable region;
add the following global declarations to the program

    struct sys_params mcall_args[K];
    int rval;

for (each clusterable region C) {
  let C consist of the sequence of system calls 'x0 = s0(ā0); ··· ; xk = sk(āk);'     [0 ≤ k < K]
  replace the code fragment C with the following code fragment:

    mcall_args[0].sys_call_no = system call number for s0;
    mcall_args[0].params = set_params(ā0);
    ...
    mcall_args[k].sys_call_no = system call number for sk;
    mcall_args[k].params = set_params(āk);
    rval = multi_call(k, mcall_args);
    if (rval ≤ k) {   /* an error occurred */
      x0 = mcall_args[0].result;
      ...
      xk = mcall_args[k].result;
      ... error handling code ...
}
```

Figure 7: Transforming a program to use multi-calls

3.3.3 Loop Unrolling

System call sequences very often occur inside a loop, where they may be separated by application code that processes—and therefore depends on—the data from the system calls. This is illustrated by the computation structure of an application such as gzip:

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

Loop unrolling can sometimes be applied in such cases to eliminate dependencies. Specifically, if we unroll the loop once and merge the footer of the current loop iteration with the header of the next iteration the dependency is eliminated. The following code segment shows the unrolled version of the program. The footer, the write statement, is prelude to the read system call, the header for the next iteration. This transformation is similar to shifting the loop window by half the loop length. Notice that this transformation eliminates the conditional dependency that exists in the original code. The loop condition remains the same and the semantics of the program do not change. However, instead of the read-write grouping, the program now has a write-read grouping, where the write call is always followed by a read, that is, the intervening test has been eliminated. The resulting code has the following structure:

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

The write-read cluster can now be replaced by a single multi-call.

3.3.4 Overall Approach

Our overall approach to multi-call optimization is as follows. We repeatedly apply the following steps until there is no change to any clusterable region:

1. Apply the algorithm of figure 6 to identify clusterable regions.
2. Apply the code transformations described in section 3.3 to create further opportunities for clustering.

The resulting clusterable regions can then be replaced by multi-calls using the algorithm of figure 7. Notice that this code uses the value returned by the multi-call to determine whether any of the system calls produced an error, as illustrated by the example in figure 3. If an error is found to have occurred, the value returned by the multi-call identifies the source of the error; in this case the return values from the individual system calls are retrieved from the mcall_args array, after which the original error-handling code is invoked.

3.4 Further Optimizations

While multi-calls reduce the number of jumps into the kernel, further optimizations may be carried out within the kernel. This is appropriate, for example, for a system vendor who desires to improve the efficiency of specific software components, e.g., a file transfer program or an MPEG player. Specifically, we can develop specialized and optimized versions of multi-calls for popular sequences of system calls. Popular sequences of system calls can be specialized in two phases. First, specialized multi-calls can be created and installed for such sequences. These new system calls are simply installed as loadable modules with unique system call numbers greater than 240. Then, these specialized multi-calls can be optimized using well-known compilation techniques. Going back to the copy example, we can now replace the multi-call with a read-write system call with the following signature:

int read_write(**int in, out fd, char* buff, int size);**

Internally the read_write system call would simply contain calls to the original file system read and write function pointers. Its parameters are a union of the parameters passed to the individual read and write calls.

Recall that the linearity requirement of section 3.2 was motivated by the fact that the general multi-call mechanism handled only straight-line sequences of system calls. This is no longer the case when we consider specialized multi-calls. When creating such specialized multi-calls, therefore, the linearity requirement of section 3.2 may be omitted.

Another optimization opportunity that is encountered in the course of such specialization occurs when the entire body of a loop in the program becomes a clusterable region. In such cases, the performance of the multi-call may be enhanced further by considering the entire loop—instead of just its body—as a multi-call, so that the entire execution of the loop can be accomplished using just one kernel boundary crossing instead of a crossing per iteration. This can be achieved by adding one more rule to the main loop of the algorithm in figure 6: *if a clusterable region C comprises the entire body of a loop in the program, then C is expanded to include the loop itself.*

Value profiles can be used for value-based specialization by analyzing frequently occurring argument values for popular multi-calls. For example, in the file copying example, value profiling reveals that the size argument almost always takes value 4096. When creating the new read_write system call, this information can be used to generate code where the critical path is optimized for the case where this argument is 4096; other values for this argument are handled by unspecialized code away from the critical path through this call [14].

Once specialized system calls are introduced into the system, the final optimization is to use various compiler optimizations to fine tune the performance. Examples of such optimizations include *code inlining*, i.e., inlining code from the constituent system call pointers; and *constant propagation*, which propagates the values of constant arguments into the bodies of new system call. These optimizations pave the way for other compiler optimizations such as strength reduction and dead and unreachable code elimination.

4 Experimental Results

4.1 Overview

This section describes the results of experiments run to test the effectiveness of the system call clustering approach on various applications. It also includes control experiments that measure the costs of system calls. The primary setup included Pentium 2-266Mhz laptops with 64 MB RAM running Linux 2.4.2-2. Other platforms used for control experiments included Solaris 7 on a 4-processor Sun Enterprise 450 with 2 GB RAM, Linux 2.4.18-rmk3 on a StrongARM-based Compaq iPAQ, Linux 2.4.4-2 on a Pentium 3-650 Mhz desktop, and Linux 2.4.0-test1-ac1-rmk7-crl2 on the StrongARM-based Compaq Personal Server prototype (*skiffs* [10]).

Optimization results and system call costs are reported in terms of execution time and, when possible, in terms of clock cycles. Intel Pentium processors provide a 64 bit

	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

hardware counter (Read Time Stamp Counter, RDTSC) that allow for cycle-level timing thought the rdtsc and rdtscl calls. rdtsc returns the whole value of the counter, while rdtscl returns the lower half of the counter. This counter is incremented with each clock tick and is commonly used for performance benchmarking. The cost of performing two consecutive rdtscl calls is 33 cycles.

4.2 Control Experiments

The first set of experiments measures the cost of switching into the kernel mode and back to the user mode on Linux running on a Pentium 2 laptop and compares this cost to the cost of performing a simple user-space procedure call. The measurement was performed by introducing a dummy system call jump_cost that takes one parameter and simply calls the rdtscl macro and returns this value. We enclose the jump_cost call within two rdtscl calls in the user program. The cost of entry into the system call is computed as the difference between the first rdtscl and the value returned by the system call. The value returned by the system call minus the following rdtscl gives the exit cost. The cost of a user-space procedure call was measured by converting the jump_cost system call into an equivalent user-space procedure.

Table 1 gives the results of the experiments. Note that the cost of rdtscl calls (33 cycles) is subtracted from the measured result. These results indicate that clustering even two system calls and replacing them with a multi-call can result in a savings of over 300 cycles every time this pair of system calls is executed.

Although system calls are conceptually similar across hardware and operating system platforms, implementations differ. The second set of experiments compare the costs of typical system calls on different platforms. Timing for this experiment was obtained by enclosing various different system calls including getpid, gettimeofday, read, and write within two gettimeofday instructions. read and write were given null parameters.

Table 2 gives the results of the experiments. The measurement overhead (column 6) was computed by executing two consecutive gettimeofday system calls and calculating the difference. Note that the measurement overhead has not been deducted from the numbers. These results indicate that the system call overhead on our primary test platform (Pentium 2 laptops running Linux) is comparable, or even lower, than the cost on many other typical platforms. This implies that system call clustering has potential for considerable improvements on these platforms as well.

4.3 Optimization Experiments

System call clustering was applied to a number of different programs on the Pentium 2 laptop. Profiling as described above provided sufficient information on a significant portion of program behavior and provided candidate system call sequences for the optimization. Not all sequences are easily optimized, however; for example, system calls within libraries such as the X-window library are more difficult to handle. Thus, the results are not optimal, but rather can be viewed as a lower bound that demonstrate the potential of this technique.

Copy. The first example is the copy program that has been used throughout the paper. Although simple, this program is representative of a larger class of programs (e.g., Web and FTP servers, gzip, compress) that could benefit from this approach. System call clustering was applied by using a specialized write_read multi-call and its looped variant. The numbers reported in figure 3 were calculated by taking the average of 10 runs on files of 3 sizes ranging from a small 80 K file to large files with size around 2 MB. The maximum benefit of this type of optimization are seen in the small and medium sized files since the time to perform disk and memory operations become dominate for larger files.

The given results are for block size 4096, which is the page size on Linux and also the optimal block size for both the optimized and unoptimized copy program. Figure 8 illustrates the effects of optimization with different block sizes ranging from 512 bytes to 16K bytes. The results with smaller block sizes show larger relative improvement since the copy program must execute a larger number of the read and write operations to copy the same file. As a result, the optimized multi-call will be executed a larger number of times and thus, the relative improvement is larger.

Media Player. The second example is the mpeg_play video software decoder [21]. Several frequent system call sequences were revealed by profiling, but upon inspection, many of these existed partially or completely

	getpid	gettimeofday	read	write	Overhead
Solaris	12	1	22	22	1
Linux-arm: iPAQ	4	4	24	24	3
Linux-arm: Skiff	42	13	71	73	11
Linux-x86: Laptop	12	4	17	12	3
Linux-x86: Desktop	7	1	7	6	1

Table 2: System call timing across various platforms (microseconds)

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 3: Optimization of a copy program with block size of 4096.

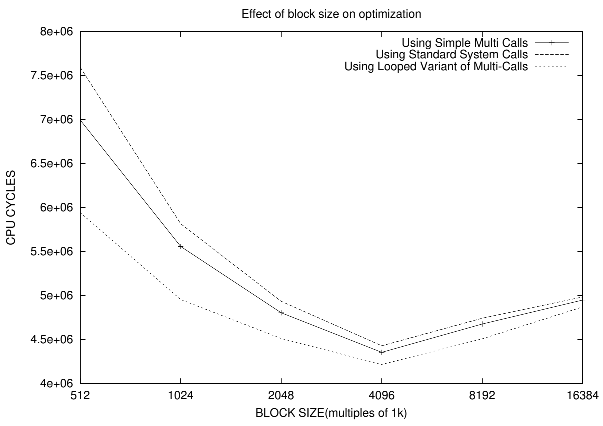


Figure 8: Effect of clustering as a function of block size.

in the X-windows libraries used by the player. Nonetheless, we were able to optimize the second most frequent sequence, consisting of a select call followed by a gettimeofday. The select system call takes 5 parameters, 4 of which always appeared as constants in the strace profile. The 5th was a pointer to struct timeval data type. The gettimeofday system call takes two parameters, the first being the pointer passed to the preceding select. The second parameter was always NULL. This sequence was replaced by a simple multi-call, which was specialized to a single select_time(struct timeval *) call. No further compiler optimizations were performed on the multi-call. Figure 4 provides the results of the optimization in terms of program execution time (seconds), frame rate (frames/second), and the number of CPU cycles required to execute the program. The program was executed using different input files with sizes varying from 4.7Mb to 15 Mb taken from [15].

Other programs. Additional experiments are currently underway, and results will be included in the final version of the paper. Specifically, system call clustering is being applied to a Web server and the Dillo Web browser for PDAs [17]. The Web server is a fully MIME compliant single threaded Web server in which a sequence of read and sendto system calls has been identified as a candidate for optimization. We also intend to repeat these tests on a StrongARM-based iPAQ, since this machine is representative of a platform where such optimization could be useful as a way to reduce power usage. Other promising optimization targets that will be explored are utility programs such as gzip and library calls such as printf that contain deterministic sequences of system calls.

5 Discussion and Related Work

5.1 Correctness Issues

Creation of a multi-call results in replacing a sequence of system calls by a single system call. In order for the transformation to be correct, the transformed program must be equivalent to the original program. All user-level transformations as described in section 3 are correct in the sense they do not introduce new side-effects. Within the kernel, the basic multi-call algorithm (figure 2) ensures that all system calls are executed in the order in which they appear in the user-level sequence and thus, the multi-call guarantees *order preservation*. An-

File	Size	Execution Time (sec)			Frame Rate (frames/sec)			CPU Cycles (10^9)		
		Orig	Opt	% Impr	Orig	Opt	% Impr	Orig	Opt	% Impr
DG-1	10.3M	98.51	78.43	20.38	27.93	35.09	25.6	51.51	41.12	20.17
DG-2	9.5M	107.71	82.60	23.31	27.47	35.82	30.4	63.65	52.09	18.17
DG-3	9.5M	59.51	44.13	25.84	27.12	36.59	34.9	31.00	21.70	30.00
DG-4	4.7M	34.50	25.87	25.01	28.14	37.54	33.4	23.75	21.74	8.47
DG-5	15.1M	111.25	82.65	25.07	28.01	37.70	34.6	60.18	52.10	13.42

Table 4: Optimization of mpeg_play

other important criterion for correctness is *error handling preservation.* User programs may respond to errors encountered during a system call. The implementation of multi-calls as described in figure 2 ensures that programs with multi-calls can replicate the error handling behavior seen in the original programs. Customizing multi-calls, which is the last phase of optimization, is also performed through correctness preserving transformations.

Use of multi-call also does not compromise existing system security and protection boundaries. Multi-calls are themselves implemented as system calls. This retains the explicit boundary between user and kernel spaces. Since the multi-call stub uses the original system call handlers, permissions and parameters are checked as in the original system. Attacks that use parameters to the multi-call such as buffer overflows can be avoided by explicit verification of handlers using routines provided by the host kernel. The general policy is to trust the user who installs the multi-call mechanism in the kernel.

5.2 Application to Power Management

In the context of small mobile devices, power is an important concern. The standard approach to power optimization has been to reduce the number of CPU cycles consumed by the program. We believe that system call clustering should yield significant energy savings. Instruction level power analysis performed as part of the μ-Amps project [13, 24] indicates that Load and Store instructions are more power intensive than other instructions. Almost all instructions in the boundary crossing path that accounts for system call overhead are either Load or Store instructions. Hence, a desirable side-effect of reducing the number of boundary crossings will be a reduction of the number of Load and Store instructions, which implies that the power saving may be even greater than the reduction in CPU cycles would imply.

5.3 Related Work

There have been several previous attempts to minimize the cost of user/kernel boundary crossing. The straightforward approach is to optimize the system call framework to reduce the overhead of each system call. Optimizations made in Linux and Solaris are good examples. Linux uses registers for parameter passing when possible to reduce the cost. Solaris includes Fast Trap system calls [12] that incur less overhead than conventional system calls by storing (and restoring) less state when the kernel functionality does not modify the original state (registers and stack). This latter restriction means that only a few system calls can utilize its functionality (see gettimeofday in Table 1). Moreover, while Fast Trap system calls are about 5 times faster than standard system calls, they are still 5 times slower than regular procedure invocations.

Another approach to optimizing system calls is the *ECalls* mechanism [16]. ECalls, a bi-directional lightweight kernel/user event delivery facility, provides a system call equivalent that does not always perform all the operations that are performed by a standard system call, such as invoking the scheduler and signal handlers before returning from the call. Similar to Fast Trap system calls, ECalls can speed up simple system calls. Other efforts have addressed the related issue of reducing the cost associated with data copying when crossing protection domains [4].

More sophisticated approaches rely on dynamic customization of system calls or their counterparts. The Synthesis Kernel [19] performs value based specialization and synthesizes new copies for frequent system calls based on runtime profiling of system calls. Synthesis, as well as all the above system call optimization techniques, optimize individual system calls and could potentially benefit further from system call clustering as described here. Conversely, some of these techniques could be used to make multi-calls more efficient.

A number of experimental operating system provide facilities for moving user-level functionality into the kernel. SPIN [2] and Synthetix [18] propose solutions based on an extensible kernel structure. Partial evaluation techniques are used to generate specialized code (called Extensions in SPIN), which can be dynamically plugged into the kernel. SPIN relies on type safety and language based features provided through the MODULA 3 programming language.

The μ-Choices operating system uses interpreted *agents* constructed as TCL scripts within the kernel to control system-level processing of multi-media streams [3]. These agents provide a mechanism that could be used to implement multi-calls. However, in contrast with multi-calls, μ-Choices uses a TCL interpreter within the kernel, something that can lead to significant performance overhead. The Jetstream system uses a similar scripting approach, but for directly controlling a device driver from a user-level protocol rather than making traditional system calls [5]. Jetstreams allows scripts of up to 64 operations consisting of 6 different device control operations to be passed through the kernel to the device driver using one ioctl system call.

Grafting kernel extensions as found in Vino [23] and *portals* in Pebble [8] are similar approaches to adding specialized system calls into a kernel. While the mechanisms provided by these experimental extensible operating systems can be used to implement multi-calls, the mechanism can also easily be implemented on conventional operating systems such as Linux. Furthermore, our clustering techniques go beyond simple mechanisms to provide a fundamental foundation for merging system calls regardless of the operating system platform.

The Exokernel operating system takes the opposite approach to these extensible operating systems. That is, rather than moving functionality into the kernel, the Exokernel approach moves the majority of the kernel functionality into user-level libraries [6]. This approach reduces the number of system calls that actually need to access the kernel and thus, reduces the system call overhead. In contrast with our approach, however, the Exokernel approach cannot be used directly in conventional operating systems.

Finally, system call profiling has been used successfully for intrusion detection and prevention [7, 22]. In [7], short patterns of normal system call sequences are collected in a program. These sequences are then analyzed offline for pattern creation and anomaly detection. More recent work performs anomaly detection online. For example, [22] uses profiles to derive patterns that represent the normal execution of a program. At runtime, system calls are intercepted and matched with good behavior. If the sequences do not match, the offending program can be terminated. This work is similar in the sense that it relies on profiling and detection of patterns, but the motivation is different.

6 Conclusions

System call clustering is an optimization approach that allows multiple system calls to be coalesced to reduce kernel boundary crossings. Multi-calls are the mechanism used to provide a single kernel entry point for a collection of calls, while execution profiling and compiler techniques are used to identify optimization opportunities and to expand their scope, respectively. Initial experimental results are encouraging, ranging up to a 20% reduction in execution time for programs such as media players that exhibit repetitive system call behavior. Improvements of this magnitude argue persuasively for the value of considering a program's entire system call behavior such as done here, not just the performance of individual calls.

Acknowledgments

P. Bridges and C. Ugarte provided excellent suggestions that improved the paper. The work of S. Debray was supported in part by NSF under grants CCR-0073394, EIA-0080123, and CCR-0113633. The work of the other authors was supported in part by DARPA under grant N66001-97-C-8518 and by NSF under grants ANI-9979438 and CCR-9972192.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] 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.

[3] 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.

[4] 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.

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

[6] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, Dec 1995.

[7] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 1996.

[8] 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.

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

[10] Compaq Cambridge Research Lab. Skiffcluster project, http://www.handhelds.org/projects/skiffcluster.html.

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

[12] 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.

[13] MIT μAMPS Project. SA-1100 instruction current profiling experiment. http://www-mtl.mit.edu/-research/icsystems/uamps/pubs/sinha_dac01.html.

[14] R. Muth, S. Watterson, and S. K. Debray. Code specialization based on value profiles. In *Proc. 7th. International Static Analysis Symposium*, pages 340–359. Springer-Verlag, June 2000.

[15] Technical University of Munich. http://www5.in.tum.de/forschung/visualisierung/-duenne_gitter.html.

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

[17] Dillo Project. http://dillo.cipsga.org.br/.

[18] 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.

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

[20] M. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. Profile-directed optimization of event-based programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, Jun 2002.

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

[22] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the 8th USENIX Security Symposium*, pages 63–78, Berkeley, CA, Aug 1999. Usenix Association.

[23] 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.

[24] A. Sinha and A. Chandrakasan. Joule-track- a web based tool for software energy profiling. In *Design Automation Conference*, Jun 2001.