Eliminating Redundant Function Calls

A Project Final Report for 15-745, Fall 2003

Indrayana Rustandi
Computer Science Department
Carnegie Mellon University
indra+@cs.cmu.edu

Jeffrey Stylos
Computer Science Department
Carnegie Mellon University
jsstylos@cs.cmu.edu

ABSTRACT

Multiple calls to the same function using the same argument values are often redundant, and the result of the earlier call can be reused. This is similar to the reuse of redundant arithmetic expressions in redundancy elimination. However, only calls that will return the same results and not produce side effects can be reused.

In this paper we present a static analysis that determines whether calls to a function can be reused. The results from this analysis are then used to create annotations that the GCC compiler can use to optimize the redundant calls. Using our analysis and compiling with GCC significantly speeds up microbenchmarks, and in many cases allows code to be written in a cleaner style without lost of performance.

1. INTRODUCTION

Redundancy elimination is a well-known technique to optimize code by reusing the result of previously computed values. Some of the redundancy elimination techniques commonly used in optimizing compilers are common-subexpression elimination, loop-invariant code motion, and partial-redundancy elimination[10]. However, all these techniques are intraprocedural in that they do not consider effects of function calls. Function calls are treated as always necessary. If there is a way to gather information about whether a function call might be redundant, then, together with standard data-flow analysis techniques, the information allows a compiler to extend redundancy elimination by removing redundant function calls.

An inherent prerequisite of knowing whether a function call is redundant is to know whether the function call generates any side-effects. This problem, the problem of interprocedural side-effect analysis, has been investigated for many years, so this is a mature area of research. In fact, there are a few algorithms that have been invented to solve the problem of interprocedural side-effect analysis ([2], [1], [11], [4], [3]). These algorithms invariably require the construction of the call graph of the program. It has been shown that, with the presence of pointers and procedure variables, which can be prevalent in a typical C program, the construction of a call graph is *PSPACE*-hard[13]. Furthermore, Myers showed that a certain type of side-effect analysis (flow-sensitive side-effect analysis) is co-*NP*-complete[11]. However, in a typical compilation of a program, the complexity of the compilation stages has to be tractable. And although some of the existing algorithms are already proven to be tractable, those algorithms are designed to compute more information than what we need.

This paper describes a stripped-down interprocedural side-effect analysis method. The contribution of this method is that it uses the side-effect information to determine whether a function might be reusable, i.e. multiple calls to this function with the same parameters will yield the same results and generate no side effects. The method does not derive from any of the existing algorithms for interprocedural side-effect analysis. Because the tractability of the method is a requirement of the method, it might be the case that it is more conservative compared to the ideal side-effect analysis, and even most of the existing algorithms. However, as will be shown later in the paper, this does not hinder the method to gather useful information to decide whether function calls might be redundant.

The other half of the picture is to use the information from the analysis to actually optimize redundant function calls. It turns out that the solution to this problem has been found and implemented in a widely used compiler. The GNU Compiler Collection (GCC)[5] accepts an extension to the C language where functions known to generate no or minimal side effects can be annotated as such. Given such an annotation, the compiler can actually determine whether a call to a certain function is redundant. However, GCC relies on the programmer to manually annotate the functions. Since we do not wish to reinvent the wheel, we focus on providing the technique to automatically annotate functions with no or minimal side effects while relying on GCC to do the optimization based on our annotations.

2. TYPES OF SIDE EFFECTS

Banning[1] formulated the side-effect analysis as computing the elements of the sets (subsequently referred to as the Banning sets):

MOD(s) the set of variables whose value may be modified by an execution of the statement s.

REF(s) the set of variables whose value may be inspected or referenced by an execution of the statement s.

USE(s) the set of variables whose value may be inspected by an execution of the statement s before being defined by the execution of s.

DEF(s) the set of variables whose value must be defined by every execution of the statement s.

The existing algorithms aim toward providing full and complete information for one or more of the above sets. However, for our method, this is always not necessary. To see why, first we introduce the GCC's classifications of functions in terms of their side effects.

By using an extension of C, GCC allows for functions to be annotated as `const` or `pure`[6]. These are two classes of functions with no or minimal side effects. `const` functions are functions that examine only their arguments. Furthermore, they have no effects besides their return value. `pure` functions, on the other hand, might potentially examine global value or value stored in memory, but they also have no effects except for their return value. So the class of `const` functions belongs as a subset of the class of `pure` functions. For convenience, we call functions that are neither `const` nor `pure` "unreusable" functions. Figure 1 gives an example of each type of functions, and figure 2 gives an example of how the applicable functions from figure 1 are annotated based on GCC's extension to C.

We can now see how `const`, `pure`, and unreusable functions relate to the types of side effects proposed by Banning. Both `const` and pure functions do not have any effects except their return value. Hence, any modification to global values or non-local memory in a function makes that function to be unreusable. Translated in terms of Banning's sets, a function must be unreusable if it contains one or more statements whose **MOD** set contains non-local variables. When we say non-local variables, we implicitly include variables whose value can be accessed (read or modified) through pointers passed to the function.

The **REF** set for a statement contains the variables whose value is referenced by the execution of the statement. Or we can also say that these variables are read through the execution of the statement. Hence, if a function contains a statement whose REF set includes non-local variables, then this function might be `pure`, but cannot be `const`. Whether the function is actually `pure` or not depends on the **MOD** set of the statements, as described in the previous paragraph.

The **USE** and **DEF** sets depend on flow-sensitive information[1]. In other words, to determine the **USE** and **DEF** sets for a particular statement in a function, we need to know the control-flow information in that function, including other function calls inside it. For this reason, it might not be clear how to annotate the function based on the **USE** and **DEF** sets. However, for our purposes, we do not need

```
int x = 0;


int square(int n) {
    return n * n;
}

int square_global() {
    return x * x;
}

void inc_global() {
    x++;
}
```

Figure 1: Examples of const (square), pure (square_global), and unreusable (inc_global) functions.

```
int x = 0;


__attribute__ ((const))
int square(int n) {
    return n * n;
}

__attribute__ ((pure))
int square_global() {
    return x * x;
}

void inc_global() {
    x++;
}
```

Figure 2: Functions in figure 1 annotated based on GCC's attribute annotations.

the information in these sets. For our analysis, we do not care whether a definition of a variable in a statement occurs by itself or whether the statement uses the value of the variable before its definition. The definition, on the other hand, implies that there is a modification to the variable. So we can put elements in the **USE** and **DEF** sets in the **MOD** set, and use the augmented **MOD** set in the same way as described above.

3. METHOD/TECHNIQUE

The relationship between the types of side effects and their relationship to classes of `const`, `pure`, and unreusable functions as described in the previous section forms the basis of the development of our method. A simple technique is to calculate all four Banning sets and determine the proper annotations following the logic in the previous section. However, as we have emphasized, the information provided by the Banning sets is not all necessary to determine the class of functions. So it might not be the best technique, especially in terms of complexity.

3.1 Analysis of Functions Containing Stores/Loads but No Function Calls

For simplicity, we begin by considering functions that do not contain function calls. Let us consider the discriminating factors between reusable (`const` and `pure`) and unreusable functions: the elements of the **MOD** set for its statements. This is a good place to start, because once we know that a function is unreusable, then it cannot be either `const` or `pure`, so we do not need to check the **REF** set of its statements. Furthermore, we do not need to compute all the elements of the **MOD** set. Any modification to non-local variables in a statement implies that the function is unreusable, so once we find one such case, the analysis can terminate with the result that the function is unreusable. Included in this kind of statement are stores to global variables and stores to aliased memory. This is a case where the full information of the Banning set is not needed.

If the function is found to be reusable through the above step, then now we need to find whether it is `const` or `pure`. In the previous section, we described that this can be done by looking at the elements of the **REF** set of the statements in the function. In our method, instead of calculating the **REF** set, we look for a statement containing load of non-local variables. Any occurrence of such a statement will mean that the function is `pure`, so we need to look only for the first occurrence. When such a statement is found, we can terminate the analysis with the result that the function is `pure`. If after going through all statements in the function without finding any statements loading non-local variables, then we can terminate the analysis with the result that the function is `const`.

3.2 Analysis of Functions Containing Function Calls but No Stores/Loads

Now we extend our method to consider functions containing calls to other functions. If we computed all the Banning sets and used the results, then function calls were not an issue in our analysis, since the information contained in the Banning sets takes into account side effects that can be generated through function calls. However, since we decide not to use any Banning set computation, we have to develop an alternative to account for function calls.

First, let us consider the case where the current function contains function calls but no store to or load from non-local variables. Let us further assume that there is only one function call in the current function. In this case, any effects that the current function has will depend on any effects that the called function has. So the current function should inherit the class of the called function. With this fact, now we consider that the current function contains multiple function calls. Which class should be assigned to the current function? The effects of the current function will be the union of the effects of all the called functions. If one of the called function is unreusable, then the current function will have effects in the form of modification to non-local variables, and by definition the current function will also be unreusable. Similarly, if none of the called functions are unreusable but there is one or more which is `pure`, then the current function will have effects in the form of read from non-local variables, which means that it has to be `pure`. Only if there are no calls to unreusable or `pure` functions—in other words, all calls are to `const` functions—will the current function be `const`. So that is how we determine the class of the current function.

	unreusable (function calls)	pure (function calls)	const (function calls)
unreusable (stores/loads)	unreusable	unreusable	unreusable
pure (stores/loads)	unreusable	pure	pure
const (stores/loads)	unreusable	pure	const

Table 1: Matrix of Aggregate Effects of Stores/Loads and Function Calls

To determine the class of a called function, we apply the same method to it. This will present no problem if the call graph from the current function is in the form of a tree. In this case, the method will traverse the call graph to the functions at the leaves and propagate the results up the tree. However, cycles in the call graph present a problem, because the method described so far might potentially loop through these cycles infinitely when it cannot determine the class of the functions in the cycle. Nevertheless, in the case when all functions in the cycle do not contain any stores or loads, all functions in the cycle can be considered to be `const`, since there are no stores or loads in the whole cycle. We will consider cases of cycle with functions that may contain stores and loads in the next subsection.

3.3 Analysis of Functions Containing Both Stores/Loads and Function Calls

So far, we have considered disjoint types of functions, i.e. functions containing stores or loads to non-local variables but no function calls, and functions containing function calls but no stores or loads. Now let us see how to analyze functions containing stores and/or loads along with function calls. Considering the three possible results from both kinds of analysis and the results from both kinds of statements, we get the 3x3 matrix drawn in table 1 for the aggregate effect on the current function. To combine the two analyses, we can do one after the other. In terms of results, there will be no difference whether we start by analyzing side effects based on stores or loads, or whether we start by analyzing side effects based on function calls. However, the decision might affect how fast we reach the results for the whole program. This will be clarified when we discuss convergence of the method in the presence of cycles next.

Consider the example in figure 3. Assume that `f1` is analyzed first. If the method considers stores and loads first, then right away `f1` will be analyzed as unreusable due to `printf`. When the method visits `f2`, because `f1` has been annotated as unreusable, then `f2` is also determined to be unreusable. This is the final result of the analysis, however, if the method considers function calls first, then analyzing `f1` will lead it to analyze `f2`. When the method reaches `f2` this way, it determines that a cycle exists, and the method will conditionally annotate both `f1` and `f2` as `const`. On the next step, when it considers the stores and loads in `f1`,

```
voidf1() {
    f2();
    printf(‘‘hello’’); /* unreusable */
}

void<= f2() {
    f1();
}

void inc_global() {
    x++;
}
```

Figure 3: A simple example to illustrate convergence in the presence of cycles.

it will change the annotation of `f1` to unreusable. Since the method already visits `f2`, it might seem that the analysis is done. However, that is not the case. We need to revisit `f2` because the attribute of the function it calls (`f1`) changes. As shown above, the method might require fewer steps by considering stores and loads first.

The above example shows how the method deals with cycles. In general, the method starts by assuming that all functions are `const`. It will then analyze each function for `pure` side effects or unreusability. Any change in the annotation of a function needs to be propagated along the call graph, including cycles. This will include changes resulting from the propagation process itself (e.g. in the above example, `f2`'s annnotation changes as a result of the propagation of the information that `f1`'s annotation changes). The process runs until there are no more changes of a function's annotation that needs to be propagated.

Although we have not done any formal analysis, we see some similarity between this problem—regardless of whether the method considers stores and loads first, or whether it considers function calls first—and a generic data-flow analysis problem[9]. The propagation of information along the call graph is similar to the propagation a data-flow information along a control-flow graph. In addition, the relationship `const`–`pure`–unreusable forms a finite descending chain from least to most conservative. By initially assuming functions to be `const` and propagating less restrictive attributes (`pure` and unreusable), the method will converge to the final result.

4. IMPLEMENTATION

We implemented our method in the SUIF2 research compiler framework[7]. Within this framework, the method works on each `ProcedureDefinition`. All annotations are appended to the `ProcedureSymbol` corresponding to the `ProcedureDefinition`. In a `ProcedureDefinition`, the method first looks for any occurrence of `StoreStatement` or any occurrence of `StoreVariableStatement` with a global variable as the destination. If such a case is found, the method annotates the procedure as unreusable and terminates. If the opposite occurs, then the method proceeds to look for any occurrence of `LoadExpression` or any occurrence of `LoadVariableExpression` with a global variable as source in the `ProcedureDefinition`. The method stores a value to remember this fact, but in both cases it does not terminate. Instead, it goes through each `CallExpression` and `CallStatement`. For each call where the symbol of the function being called is available (i.e. not a call through function pointers), the method checks the annotation for the corresponding `ProcedureSymbol`. If the `ProcedureSymbol` contains no annotation for results, then we run the method on its `ProcedureDefinition`. Functions containing no definition (i.e. functions defined in other compilation unit or available in a library) are considered unreusable. We refer to the previous section for how we determine the attribute of the currently analyzed function based on both the stores or loads and function calls.

To enable our analysis results to be passed to GCC, we have to convert the SUIF representation to C. In particular, we need to modify the `s2c` program that comes in the SUIF2 distribution so that the SUIF annotations are converted properly to the GCC attribute annotations, similar to the example in figure 2. This involves modifying the macro file defining the rules for converting SUIF representation to C (`nci/suif/suif2b/basesuif/s2c/c_text.mac` in a standard SUIF2 distribution).

5. RESULTS

Our analysis by itself will not give any performance improvement. But when the result of the analysis is compiled by GCC, some performance gain might be obtained in the program compiled, especially when it contains function calls that can be optimized because they are redundant. To measure this performance improvement, we wrote four microbenchmarks[1]:

benchmark1 loop for one million iterations; the body of the loop contains two sequential calls to a `const` function (`square`, containing multiplication), both with the same arguments.

benchmark2 loop for one million iterations; the body of the loop contains two sequential calls to a `pure` function (`mult_with_global`, containing multiplication), both with the same arguments, and where the global value is changed in each iteration.

benchmark3 loop for one million iterations; the body of the loop contains two basic blocks, each containing call to a `const` function (`divide_by_two`, containing division), and both calls have the same arguments.

benchmark4 loop where the loop condition contains a call to a `const` function (`square`).

All the benchmarks were written in C. We created two versions of each benchmark: an original version of the benchmark without our analysis, and a modified version with our analysis. To compile the benchmark to obtain the modified (analyzed) version, we first compiled the C source file

[1]sources for the microbenchmarks are available at `http://www-2.cs.cmu.edu/%7Eindra/15745/benchmark.tar.gz`

	original (ms)	modified (ms)	speedup
benchmark1	12	8	1.5
benchmark2	13	13	1.0
benchmark3	11	7	1.57
benchmark4	679	130	5.22

Table 2: Benchmark Results

of the corresponding benchmark to its SUIF representation, using the `c2s` program that comes in a standard SUIF2 distribution. Our analysis pass would then be applied to the resulting SUIF representation, and the result of our analysis would be converted back to C using SUIF's `s2c` pass. Similarly, to obtain the original version, we took the above steps with the exception of running our analysis. So we converted the C source file of the benchmark to SUIF using `c2s`, and converted the SUIF representation back to C using `s2c`. One might ask why these steps are necessary for the original version instead of using the original C source file for the benchmark. The reason we did this is because the `s2c` pass might produce a different, although similar, C representation from the original C source file, and we want to reduce any variability that this might cause.

Both C files produced by `s2c` were then compiled using GCC, with the options `-O3` and `-fno-inline`. The option `-O3` allows GCC to optimize calls to functions with `const` or `pure` attribute. However, `-O3` will also enable function inlining in GCC, so we disable inlining using the option `-fno-inline`. We disabled inlining for our benchmarks because we are trying to measure the performance of function calls. We used GCC 3.2.2 on a Red Hat Linux 9 machine. This step will produce executables for both the original version and the modified version of the benchmarks. We then ran each version of each benchmark ten times on a Pentium IV 2.4 GHz machine with 1GB of RAM running Red Hat Linux 9. Table 2 contains the results of the benchmark runs, using the averaged user time as measured by the UNIX utility `time` over the ten iterations.

As the table shows, there are speedups gained in benchmarks 1, 3, and 4, all of which contain `const` functions. However, for benchmark 2, which contains a `pure` functions, we did not get any speedups. We were a bit surprised ourselves by the results for benchmark 2. So we inspected the assembly code produced by GCC for the executable of benchmark 2, and indeed we found out that although the second call to the `pure` function is redundant because there is no change to the global variable in between the calls, GCC somehow does not optimize this case. GCC might not handle optimization for `pure` functions yet. Regardless, at least for the case of `const` functions, we have shown that we can get significant performance improvement by optimizing redundant calls. The performance improvement does depend on the complexity of the function being called. So the effect of the analysis and optimization will be more pronounced when there are a lot of redundant calls to computationally expensive but `const` functions (e.g. functions for common mathematical operations).

One might also ask what improvement our analysis combined with GCC when it is used to compile standard benchmark programs, for instance, the programs that are part of the applicable SPEC benchmark suite[8]. A limitation in our implementation prevents us to do this. This limitation—the inability of our implementation to analyze over multiple compilation units—will be discussed in the next section.

6. LIMITATIONS AND FUTURE WORK

There are several limitations which we will now describe. These limitations also illustrate improvements that can be done in the future.

6.1 Multiple Compilation Units

Currently, our implementation is unable to analyze calls to functions which are defined in another compilation unit (e.g. in a different source file). When the method encounters a case of this, it will treat the functions being called as unreusable, which potentially makes the results to be too conservative.

A solution to this problem can be developed by first dividing the functions defined in a different compilation unit into two categories:

1. Functions defined in a different compilation unit that will be linked to a single executable during runtime.
2. Functions defined in a dynamically loaded library.

We can extend our method to account for the first category of functions by doing partial analysis during compilation of a single compilation unit, and doing full analysis during link time. This is similar to the method of constructing call graph for functions in separate compilation units described in [10]. However, there are details of this method that will need to be worked out, which are beyond the scope of this paper.

The second category of functions present a more difficult case. In most cases, the programmers do not even have access to the source of functions defined in a dynamically loaded library. And since these functions are resolved fully only during the execution of the program, even if they are annotated as such, the optimization based on their annotations might be possible only during runtime. This is within the scope of dynamic compilation. And even if we can obtain the information when the functions are resolved partially during link time, we might still need to modify the executable/library format to accommodate for the annotations.

6.2 Static Properties of Functions

Our analysis can detect two static properties of a function: being `pure`—having no side effects (for any argument values to the function)—or `const`—having no side effects and also not depending on global variables or values pointed to by pointer arguments (for any arguments). However, there may be functions that sometimes produce side effects (or depend on values in memory) but do not for some classes of input. For example, a "safe-divide" function that divided

```
int square(int n) {
    int i;
    int *ip = &i;
    *ip = n * n;
    return i;
}
```

Figure 4: An example of a case where pointer analysis can help obtain less conservative results.

two integers or printed an error if the divisor were zero would be reusable (`const`) when the second argument was known to be non-zero. Our analysis will not detect these and GCC is not able to optimize calls to these functions.

Overcoming this limitation would require analysis for each function call rather than each function, would be more expensive and would not allow the called function to be analyzed separately ahead of time.

6.3 Pointers

Our method already handles pointers in a basic way. Nevertheless, doing a more comprehensive pointer analysis (for instance, using the method described in [12]) might yield information that can be used to obtain more optimistic results. For instance, consider the example in figure 4. When analyzed using our existing method, the function `square` will be annotated as unreusable because there is a modification to a location pointed by a pointer (`ip`). However, if we consider that the pointer `ip` points only to a local variable within the function, `square` is really a `const` function.

Another aspect of pointers that we have not dealt with is the concept of function pointers. Currently we handle only function calls using the symbol of the functions being called. Any calls through function calls cause the current function to be unreusable. With pointer analysis, it might be possible in some cases to determine the identity of the function called by way of a pointer.

6.4 The Halting Problem

The semantics of `pure` and `const` that we use are slightly different from those defined by GCC. GCC specifies that both `pure` and `const` functions must terminate. In our method, we annotate the procedures without trying to prove termination. This could potentially cause problems if GCC hoisted a call to a non-terminating function so that it was called before it would have been. (We do not know if GCC actually does this, but the specifications of `pure` and `const` would seem to allow it.)

To prevent this, we could be conservative and only annotate functions that we could prove will terminate. Alternatively, the GCC optimization pass could be modified so that it never moved a function call to an annotated function above side effects or earlier than it is anticipated.

Because this could potentially cause incorrect results (with non-terminating functions), it would be important to resolve this issue before adopting it in a real system.

7. CONCLUSION

We have shown that automatically detecting the reusability of functions is feasible and works in conjunction with the optimization that the GCC compiler can perform on such functions. We have also shown that there is promising performance improvement that can be gained through optimization of reusable functions. To conclude, we describe yet another benefit that such analysis and optimization can provide. If they are available in commercial compilers, then the analysis and optimization might potentially relieve the programmers of the burden of optimizing function calls manually. This might also lead to more readable, and hence maintanable code.

8. ACKNOWLEDGMENTS

We are grateful for the comments and suggestions given by Todd Mowry and David Koes throughout the duration of the project. In particular, we found out about the `const` and `pure` attributes for functions in GCC through communication with David Koes.

9. REFERENCES

[1] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41. ACM Press, 1979.

[2] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, 21(9):724–736, 1978.

[3] D. Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 47–56. ACM Press, 1988.

[4] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 57–66. ACM Press, 1988.

[5] http://gcc.gnu.org. The GNU Compiler Collection.

[6] http://gcc.gnu.org/onlinedocs/gcc/Function Attributes.html. Declaring Attributes of Functions.

[7] http://suif.stanford.edu/suif/suif2/index.html. The SUIF 2 Compiler System.

[8] http://www.spec.org. SPEC – Standard Performance Evaluation Corporation.

[9] T. Marlowe and B. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[10] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[11] E. M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–230. ACM Press, 1981.

[12] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM Press, 1996.

[13] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–94. ACM Press, 1980.