

# Eliminating Redundant Function Calls

## A Project Proposal for CS 745, Fall 2003

Jeffrey Stylos, Indrayana Rustandi

{ jsstylos, indra+ }@cs.cmu.edu

<http://www.cs.cmu.edu/~jsstylos/15745/>

### ABSTRACT

In this paper we propose a project to design and implement a compiler optimization that will optimize function calls by reusing the result of a previous function call, when it can be proven statically that the two calls will have the same result and not produce side effects. We describe this optimization and present a detailed plan on how we will implement it.

### PROJECT DESCRIPTION

Partial redundancy elimination (PRE) is a well known technique to optimize code by reusing the result of previously computed values. However, it is limited in the types of redundant expressions it can eliminate. We propose to extend partial redundancy elimination to eliminate redundant function calls by reusing previous function call results. This will provide the benefits that inlining and PRE would provide together, but be applicable in situations when inlining is either not possible or not desired.

### Motivation

In addition to creating better code, this optimization will enable programmers to write more readable code. Instead of explicitly creating a temporary to store a function call result and using the temporary, performance conscious programmers will be able to call functions multiple times, safe in the knowledge that the compiler will optimize away redundant calls. In many cases, this will enable smaller, more readable source code with fewer variables.

### Limitations

Function call results can only be reused if the function does not produce and side-effects and if its result can be proven to be the same for two calls. Showing that a function will return the same result given the same arguments is easy for simple integer or floating point arguments but more complicated when dealing with pointers. Instead of just knowing whether the arguments are the same, we have to know if all the memory that will be loaded by the function through the pointer will be the same. This can be difficult, especially when dealing with pointers to pointers, or objects such as strings, whose size might not be known statically. For this reason, we propose a conservative initial approach that deals only with functions with non-pointer arguments. We will then extend this to work in more situations.

### Context

We will implement our optimization inside the SUIF research compiler for C.

### LOGISTICS

#### Plan of Attack and Schedule

We plan to complete our project in two main stages: first, we will implement an optimization that reuses the results of functions that do not have pointer arguments. Second, we will extend the optimization to work in some situations when the function does have pointer arguments.

Our initial, non-pointer, optimization has two passes. First, it will perform an analysis of each function being compiled, annotating it with whether or not it creates any side-effects and if it is a candidate for call reuse. Second, it will analyze code looking for calls to functions with the same argument values. If the function has been annotated as safe to reuse, then the later function calls will be replaced with copies of the initial return value.

Our tentative week-by-week schedule is as follows. (In each of the programming tasks, we plan on using the same collaborative co-writing approach to programming that we used for assignments 2 and 3. For the report writing tasks, we plan to divide the paper into sections that will be written separately, and have one person in charge of combining and finalizing each report.)

#### Week 1 (Oct 27 – Nov 2)

- Submit proposal
- Set up compilation environment
- Study literature

#### Week 2 (Nov 3 – Nov 9)

- Design and pseudocode procedure analysis
- Start implementing procedure analysis

#### Week 3 (Nov 10 – Nov 16)

- Finish implementing procedure analysis
- Design and pseudocode redundant call removal
- Start implementing redundant call removal

#### Week 4 (Nov 17 – Nov 23)

[*Milestone week*]

- Finish implementing redundant call removal
- Write project milestone report
- Design a more general optimization that deals with some pointer cases

#### Week 5 (Nov 24 – Nov 30)

- Start implementing more general optimization

#### Week 6 (Dec 1 – Dec 7)

- Finish implementing more general optimization
- Write project summary

#### **Milestone**

By 11:59pm (EST) on November 19<sup>th</sup> 2003, we plan to have implemented an optimization to reuse the results of function calls for functions without pointer arguments.

#### **Literature Search**

Our optimization involves various kinds of interprocedural analyses. We have started our search with chapter 19 of Muchnick textbook [1], which deals with interprocedural analyses and optimizations. In particular, we will analyze a particular procedure for side-effects and aliases. The above chapter describes the issues and the various techniques that have been discovered for these kinds of interprocedural analysis. Following the citations in the Muchnick textbook we have found papers by Banning [2], Myers [3], Callahan [4], Cooper & Kennedy [5], and Deutsch [6], which seem to be relevant to the interprocedural analysis that we need. These papers describe original techniques to do flow-sensitive and flow-insensitive side-effect and alias analyses.

Within the context of the six kinds of interprocedural optimizations described in section 19.5 of the Muchnick textbook, our approach roughly belongs to the sixth kind:

using interprocedural data-flow analysis to intraprocedural optimizations. We are not able to find papers that describe our proposed elimination of redundant procedure calls, and we believe this to be novel research.

#### **Resources Needed**

We plan on using our existing SUIF/MachSUIF compilation environment under Linux.

#### **Getting Started**

Though we have not started this project beyond searching the literature and writing this proposal, we are ready to start immediately.

#### **ACKNOWLEDGMENTS**

We wish to thank Dave Koes and Todd Mowry for helping us turn our idea into a feasible project, and also for the generous final project grade they are sure to give us.

#### **REFERENCES**

1. Muchnick, S.S., *Advanced Compiler Design and Implementation*. 1997, San Francisco, CA: Morgan Kaufmann.
2. Banning, J.P., *An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables*. POPL, 1979: p. 29-41.
3. Myers, E.W., *A Precise Inter-procedural Data Flow Algorithm*. POPL, 1981: p. 219-230.
4. Callahan, D., *The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis*. PLDI, 1988: p. 47-56.
5. Cooper, K.D. and K. Kennedy, *Interprocedural Side-Effect Analysis in Linear Time*. PLDI, 1988: p. 57-66.
6. Deutsch, A., *Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting*. PLDI, 1994: p. 230-241.