

**OTIMIZAÇÕES DE CÓDIGO SENSÍVEIS A
CONTEXTO BASEADAS EM CLONAGEM DE
FUNÇÕES**

MATHEUS SILVA VILELA

**OTIMIZAÇÕES DE CÓDIGO SENSÍVEIS A
CONTEXTO BASEADAS EM CLONAGEM DE
FUNÇÕES**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte
Fevereiro de 2014

MATHEUS SILVA VILELA

**CONTEXT-AWARE CODE OPTIMIZATIONS
BASED ON FUNCTION CLONING**

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais - Departamento de Ciência da Computação in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

February 2014

© 2014, Matheus Silva Vilela.
Todos os direitos reservados.

Ficha catalográfica elaborada pela Biblioteca do ICEX - UFMG

Vilela, Matheus Silva

V699c Context-aware Code Optimizations Based on
Function Cloning / Matheus Silva Vilela. — Belo
Horizonte, 2014

xviii, 68 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais - Departamento de Ciência da
Computação

Orientador: Fernando Magno Quintão Pereira

1. Computação - Teses. 2. Arquitetura de
computador. 3. Compiladores (Computadores).
4. Clonagem de código. I. Orientador. II. Título.

CDU 519.6*21(043)



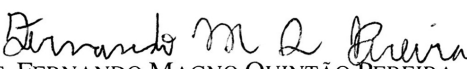
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

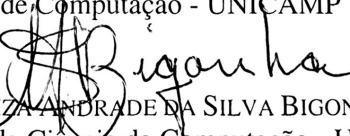
Otimizações de código sensíveis a contexto baseadas em clonagem de funções

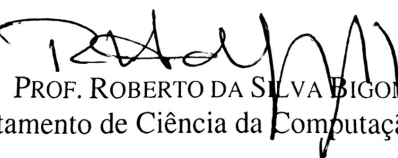
MATHEUS SILVA VILELA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. EDSON BORIN
Instituto de Computação - UNICAMP


PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG


PROF. ROBERTO DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 21 de fevereiro de 2014.

Resumo

Compiladores fazem uso de duas técnicas principais para implementar otimizações dependentes do contexto de chamadas de funções: integração e clonagem. Historicamente, a integração de funções tem tido um uso mais amplo, já que, na prática, tende a ser mais efetiva. Apesar disso, a clonagem de funções provê benefícios não presentes na integração. Em particular, a clonagem dá ao desenvolvedor uma maneira de mitigar *bugs* de desempenho, já que sua aplicação gera código reusável. Além disso, ela lida com recursão de uma forma mais natural. Finalmente, a clonagem pode levar a uma menor expansão de código.

Neste trabalho, revisamos a clonagem de funções sob a luz desses benefícios. Discutimos algumas técnicas de especialização de código independentes baseadas em clonagem que, apesar de simples, são amplamente aplicáveis, mesmo em *benchmarks* altamente otimizados, como o SPEC CPU 2006. Nossas otimizações são de fácil implementação e aplicação. Utilizamos a conhecida heurística de *profiling* estático de Wu e Larus para medir o ganho de desempenho de um clone. Essa métrica nos dá uma maneira concreta de indicar, aos desenvolvedores de código, *bugs* de desempenho potenciais, além de nos fornecer uma métrica para decidirmos se devemos manter ou não um clone. Ao implementarmos nossas ideias no compilador LLVM, fomos capazes de obter ganhos de desempenho significantes nos *benchmarks* do SPEC, mesmo que aplicados sobre o nível de otimização -O2.

Palavras-chave: clonagem, LLVM, otimizações sensíveis ao contexto.

Abstract

Compilers rely on two main techniques to implement optimizations that depend on the calling context of functions: inlining and cloning. Historically, function inlining has seen more widespread use, as it tends to be more effective in practice. Yet, function cloning provides benefits that inlining leaves behind. In particular, cloning gives the program developer a way to fight performance bugs, because it generates reusable code. Furthermore, it deals with recursion more naturally. Finally, it might lead to less code expansion, the inlining’s nemesis.

In this work, we revisit function cloning under the light of these benefits. We discuss some independent code specialization techniques based on function cloning, which, although simple, find wide applicability, even in highly optimized benchmarks, such as SPEC CPU 2006. We claim that our optimizations are easy to implement and deploy. We use Wu and Larus’s well-known static profiling heuristic to measure the profitability of a clone. This metric gives us a concrete way to point out to program developers potential performance bugs, and gives us a metric to decide whether we keep a clone. By implementing our ideas in LLVM, we have been able to achieve significant speed up on the SPEC benchmarks on top of the -O2 optimization level.

Palavras-chave: cloning, LLVM, context aware optimizations.

List of Figures

3.1	A program that illustrates three different clone-based optimizations.	13
3.2	The effects of clone-based constant propagation. (a) Code after constant propagation. (b) Code after loop unrolling. (c) Code after instruction folding.	14
3.3	The effects of the elimination of unused return values. (a) Code after elimination of the return statement. (b) Code after dead code elimination.	15
3.4	Program on figure 3.1, after inline expansion and dead code elimination.	16
3.5	Example that benefits from function fusion.	17
3.6	(a) Example of Figure 3.5, after preliminary merging. (b) Inline expansion. (c) Elimination of redundant parameters. (d) Replacement of clones in call sites.	18
3.7	A simple vector multiplication routine.	19
3.8	The same vector multiplication routine, with its arguments modified with the <code>restrict</code> keyword, meaning that they never alias.	20
3.9	Example in which the potential aliasing between pointers can prevent code optimizations.	21
3.10	Optimized version of <code>copy</code> , the function seen in Figure 3.9.	22
3.11	The effects of the elimination of dead stores.	23
3.12	(a) The control flow graph of function <code>divMod</code> , originally seen in Figure 3.1, augmented with probability of edge being taken. (b) Cost of each instruction in the CFG, and edge frequencies. (c) Total cost of function <code>divMod</code> . (d) CFG of <code>divMod_clone_urve</code> , seen in Figure 3.3. (e) Total cost of function <code>divMod_clone_urve</code>	25
3.13	Errors of Wu and Larus’s static profiler on SPEC CINT 2006.	26
4.1	Transformation pipeline that we use to apply function cloning.	30
4.2	Constant Propagation: Applicability on Test-Suite SingleSource.	32
4.3	Elimination of unused retvals: Applicability on Test-Suite SingleSource.	33
4.4	Pointer Disambiguation: Applicability on Test-Suite SingleSource.	33

4.5	Function Fusion: Applicability on Test-Suite SingleSource.	34
4.6	Constant Propagation: Applicability on Test-Suite MultiSource.	35
4.7	Elimination of unused retvals: Applicability on Test-Suite MultiSource. . .	35
4.8	Pointer Disambiguation: Applicability on Test-Suite MultiSource.	36
4.9	Function Fusion: Applicability on Test-Suite MultiSource.	36
4.10	Elimination of Dead Stores: Applicability on Test-Suite MultiSource. . . .	37
4.11	Constant Propagation (Sec 3.2). Func : number of functions. Clone : number of clones. Orph : number of orphan functions. Calls : number of function calls. F-ful : number of fruitful calls. F-less : number of fruitless calls. AvgP : average profit of optimized clone. Rec : number of recursive clones.	38
4.12	Elimination of unused return values (Sec 3.3).	39
4.13	Function Fusion (Sec 3.4).	40
4.14	Pointer Disambiguation (Sec 3.5).	41
4.15	Elimination of Dead Stores (Sec 3.6).	41
4.16	Maximum proportional profits obtained on SPEC CPU 2006.	42
4.17	Compilation time of our optimizations when applied to Test-Suite Single-Source.	43
4.18	Runtime variations on Test-Suite SingleSource. (a) Constant propagation. (b) Elimination of unused return values. (c) Function fusion. (d) Pointer disambiguation.	44
4.19	Size increase on binaries in Test-Suite SingleSource.	45
4.20	Compilation time of our optimizations when applied to Test-Suite Multi-Source.	45
4.21	Runtime variations on Test-Suite MultiSource. (a) Constant propagation. (b) Elimination of unused return values. (c) Function fusion. (d) Pointer disambiguation. (e) Elimination of dead stores.	47
4.22	Size increase on binaries in Test-Suite MultiSource.	48
4.23	Compilation time of our optimizations when applied to SPEC CPU 2006. .	48
4.24	Runtime variations on SPEC CPU 2006 (a) Constant propagation. (b) Elimination of unused return values. (c) Function fusion. (d) Pointer disambiguation. (e) Elimination of dead stores.	49
4.25	Runtime improvements of all our clone-based optimizations versus LLVM -O2	50
4.26	Constprop: Fruitful sites	51
4.27	Elim. of Unused Retvals: Fruitful sites	51
4.28	Function Fusion: Fruitful sites	51
4.29	Pointer Dis.: Fruitful sites	51

4.30	Elimination of Dead Stores: Number of fruitful sites	52
4.31	Maximum number of replaced call sites	53
4.32	Size increase on binaries in SPEC CPU 2006.	54
4.33	Size reduction due to our optimizations.	54
5.1	Fft function present on Stanford Benchmarks.	58
5.2	Function Fft optimized with Pointer Disambiguation.	59
5.3	dmxpy function present on Linpack Benchmark.	60
5.4	dmxpy function optimized with Pointer Disambiguation.	61
5.5	readULONG function present on povray benchmark.	62
5.6	readULONG function optimized with Elimination of Unused Retvals.	62

Contents

Resumo	ix
Abstract	xi
List of Figures	xiii
1 Introduction	1
1.1 Context	1
1.2 Contributions	2
1.3 Results	3
1.4 Outline	4
2 Background and Related Work	5
2.1 LLVM	5
2.2 Performance Bugs	5
2.3 Code Specialization	6
2.4 Context-aware optimizations	7
2.4.1 Function Cloning	8
2.5 Alias analysis	8
2.6 Optimizations	9
3 Clone-Based Optimizations	11
3.1 This Work’s Vocabulary	11
3.2 Clone-Based Constant Propagation	12
3.3 Elimination of Unused Return Values	14
3.4 Function Fusion	17
3.5 Pointer Disambiguation	19
3.6 Elimination of Dead Stores	22
3.7 Estimating the Profit of a Clone	24

3.7.1	Validating the static profiler	26
4	Experiments	29
4.1	The Cloning Pipeline	29
4.2	The Applicability of our Optimizations	31
4.2.1	Test-Suite SingleSource	31
4.2.2	Test-Suite MultiSource	34
4.2.3	SPEC CPU 2006	37
4.3	Effectiveness: Time and Space	42
4.3.1	Test-Suite SingleSource	43
4.3.2	Test-Suite MultiSource	45
4.3.3	SPEC CPU 2006	48
4.4	Discussion	54
5	Case Studies	57
5.1	The Fft Function	57
5.2	The dmxpy function	59
5.3	The readULONG Function	60
6	Final Remarks	63
6.1	Conclusion	63
6.2	Future Work	63
	Bibliography	65

Chapter 1

Introduction

This dissertation is the result of two years of research on clone-based optimizations. Our findings are summarized in the five chapters that constitute this dissertation. In the first chapter we explain our motivations, goals and main contributions.

1.1 Context

In computer science, program optimization or software optimization is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources [Sedgewick, 1984]. In the context of compilers, an optimization can not change the program output or its side effects. The only difference between an optimized program and the original one, from the user's perspective, is that the former runs faster than the latter. The compiler optimization area has been studied for a long time. The optimizations implemented on modern compilers vary from simple modifications on basic blocks to more complex changes that rely on whole program analysis.

Code reuse is one of the main forces behind the astounding growth that the software industry has experimented in the last decades [Krueger, 1992]. Testimony of this statement is the importance that industry and academia ascribe to practices such as component oriented programming, data encapsulation and interfaces as contracts [Holmes and Walker, 2013]. However, software must be reused with discipline to avoid *performance bugs*. According to Jin *et al.* [Jin et al., 2012], a performance bug is a mistake that might impact the efficiency of the program, without causing observable errors. Jin *et al.* argue that one of the core reasons behind this kind of problem is software reuse. Programmers often reuse a software component, looking for a specific functionality, oblivious to the fact that this module may perform several other tasks which are unnecessary in that particular context. We add another reason to perfor-

mance bugs: the natural evolution of programming languages. As an example, we can cite the `restrict` modifier, introduced in C99. [ISO, 1999, pp. 110]. This modifier can be used by the the developer to tell the compiler that one or more pointers, used as actual arguments of a function call, don't alias each other. Using this information, the compiler is able to apply more aggressive optimizations on the target program. However, as the `restrict` modifier has been created on the late 90's, programs written before that could not benefit from it. Furthermore, the habit and ignorance make this keyword unpopular between C programmers.

Classic optimizations cannot normally remove performance bugs. These optimizations don't take into account that the results produced by a snippet may be useful on some contexts and irrelevant on others. In order to remove this kind of bugs, an optimization may be context-aware.

The compiler literature describes two main ways to enable context-aware code optimizations: function inlining and function cloning. The former technique is much more adopted than the latter. Many industrial compilers, such as gcc, LLVM, Open64, Jikes and Mozilla's IonMonkey implement extensive inlining of functions at higher optimization levels. Function inlining supports context sensitive optimizations more effectively than cloning, because it integrates the body of procedures directly on the place where they are invoked. On the other hand, cloning provides a few advantages when compared to inlining. Firstly, a clone is a reusable unit of code, which can be invoked at different sites within or outside the optimized program. Secondly, function cloning deals more easily with recursion, as this kind of functions can not be proper inlined [Keith D. Cooper, 2012, pg.458]. Finally, cloning tends to lead to less code expansion than inlining, depending on how many times the replicated function is called. These benefits have motivated us to revisit context-aware optimizations based on cloning, a strategy that the literature has often neglected.

1.2 Contributions

Under the light of the benefits provided by cloning, we have defined and tested a code optimization methodology based on this technique. We have designed and implemented five different optimizations. One of them, constant propagation, is well-known; the other four have not been described before. We use clone-based constant propagation as a baseline for comparisons, as it has been described previously in the literature [Metzger and Stroud, 1993]. The four other optimizations that we describe are:

- *Unused return elimination*: this optimization removes from the procedure body

all return statements and every computation used to build the expression that the function returns.

- *Dead store elimination*: this optimization removes writes to memory positions that are either overwritten after the function returns, and before being read, or that are never read.
- *Function fusion*: this optimization merges a function g into the body of a function f , if the invocation pattern $f(\dots, g(\dots), \dots)$ is common enough.
- *Pointer disambiguation*: this optimization marks parameters of functions as no-aliases whenever possible, via the `restrict` keyword available in the C99 standard.

In addition to these optimizations, we define a method to measure the profitability of a clone. We rely on the static profiling heuristics proposed by Wu and Larus in the early 90's [Wu and Larus, 1994]. This estimate of the benefit of a clone gives the compiler the subsidies to decide whether it keeps a clone, or throw it away. And more importantly, this metric provides to the program developer hints about potential performance bugs.

1.3 Results

We have implemented our ideas in the LLVM compiler framework [Lattner and Adve, 2004]. This infra-structure has allowed us to test our optimizations in a vast benchmark suite, with over 4.3 million lines of code. We have found many opportunities to apply each of our four optimizations. As an example, 33% of the functions defined within the programs of SPEC CPU 2006 benefit from pointer disambiguation. Our optimizations yield non-trivial benefit on top of highly optimized programs, such as those available in SPEC, and can greatly improve the runtime of less tuned code. We have been applying our optimizations in open source audio encoding programs, with very positive results. We have found that most of the functions in these programs do not employ C's `restrict` keyword, for instance, even though its use is obviously safe. Thus, one of the main benefits of the combination of static profiling and cloning is to aid developers at the software engineering level.

More than performance numbers, however, we believe that the main contribution of this work is to bring new attention to an important compilation technique, which has been neglected both by the industry, and by the academia. A few compiler textbooks, namely Kennedy's [Kennedy and Allen, 2002, pp.594] and Grune's [Grune

et al., 2012, pp.325] approach this form of code specialization, yet on a very superficial way. The most extensive discussion about clone-based optimizations that we are aware of can be found in Mary Hall’s PhD dissertation [Hall, 1991, Cp.5]. However, even though the technique has been described before, it has yet to find space in industrial-strength compilers. In the words of Allen and Kennedy, “The Convex Applications Compiler [Metzger and Stroud, 1993] is the only commercial compiler we know that performs this optimization”. Today, this family, albeit modest, has some new members. Both `gcc`, and `Open64` can clone a function, if it is invoked only once, and receives a constant as a parameter. This is a restricted version of Metzger’s inter-procedural constant propagation. In this work, we explore cloning in a much more extensive way.

1.4 Outline

The remainder of this dissertation is organized as follows:

- **Chapter 2, Background and Related Work:** this chapter describes important concepts related to this work and a literature review.
- **Chapter 3, Clone-based Optimizations:** this chapter describes the optimizations we have implemented on top of LLVM. We discuss how we determine if a given function may benefit from a clone-based optimization, and give some simple examples to ease understanding.
- **Chapter 4, Experiments:** this chapter describes how our proposed optimizations behave when applied on well-known benchmarks. We show our results in terms of applicability, runtime and code expansion, and discuss each of these points.
- **Chapter 5, Case Studies:** in this chapter, we study some interesting cases we have found on the benchmarks. We analyze how our clone-based approach is able to improve the generated code and why that happens.
- **Chapter 6, Final Remarks:** this chapter concludes this work. We give our final remarks and discuss future work.

Chapter 2

Background and Related Work

This chapter gives an overview of key concepts related to this dissertation. It also presents a literature review, describing previous works on the area.

2.1 LLVM

The LLVM project is a collection of modular and reusable compiler and tool-chain technologies. It has an infrastructure designed for compile-time, link-time, run-time, and “idle-time” optimization of programs written in arbitrary programming languages. It was first described by Lattner and Adve [2004]. Nowadays, LLVM is used on many companies, including Google and Apple, and has become very popular on compiler research.

LLVM optimizer can take intermediate representation code from a compiler and emit an optimized version of it. This optimized version can then be converted and linked into machine-dependent assembly code for a target platform. Being open-source, LLVM provides an easy and modular way to insert new optimizations into its pipeline. It also defines a simple low-level language with strictly defined semantics used as intermediate representation. All the optimizations are applied on programs in this form.

The optimizations on this work are implemented as modular passes on the LLVM optimizer.

2.2 Performance Bugs

The main inspiration for this work comes out of two recent papers, by Chabbi *et al.* [Chabbi and Mellor-Crummey, 2012] and Jin *et al.* [Jin et al., 2012]. These two

groups present empirical evidence of the widespread occurrence of performance bugs. Chabbi *et al.* analyze, via profiling, the impact of redundant memory accesses in the performance of programs. Jin *et al.* present an extensive review of performance bugs reported in major open source projects, and discuss ways to avoid them.

In a previous work, Jovic *et al.* [2011] have shown that profiling alone may not be enough to detect performance bugs. According to them, the routines that are critical to the user’s temporal perception tend to be called only a few times in interactive applications. Nistor *et al.* [Nistor *et al.*, 2013] have developed a tool that detects repeated access to the same memory positions. This tool has effectively revealed many hidden efficiency faults on industrial software.

2.3 Code Specialization

Specialization refers to the translation of a general function into a more limited version of it. For instance, a function with two arguments can be transformed into an one-argument function by fixing one of its inputs to a particular value. In computer science, transforming a program block into a specialized version of it is called *program specialization* or *partial evaluation* [Jones *et al.*, 1993]. This kind of optimization happens when the compiler produces code specialized to certain inputs or contexts.

Code specialization can be done either at compile-time or run-time [Consel and Noël, 1996], although this classification can appear with the names static and dynamic specialization, respectively, in recent works [Grant *et al.*, 2000; Shankar *et al.*, 2005]. Using this name convention, our approach is entirely static: no check is necessary at runtime to determine which version of a clone to invoke. However, dynamic code specialization is gaining growing attention within the programming language’s community [de Assis Costa *et al.*, 2013; Samadi *et al.*, 2012; Tian *et al.*, 2011].

In the world of just-in-time compilers, de Assis Costa *et al.* [2013] have proposed to generate specialized routines based on the runtime value of the arguments passed to JavaScript functions. These values can be easily inspected, because code is being generated while the program executes. This approach resembles the constant propagation that we have recalled in Section 3.2, although it is done on-the-fly.

An adaptive compiler produces code that is specialized to a certain type of inputs. As a recent example, Samadi *et al.* [2012] have proposed an adaptive compiler for CUDA. This compiler generates code with a switch that contains different routines to handle different kinds of inputs. The runtime value of the particular input determines which routine will be activated. Compared to the techniques that we proposed in this

paper, the adaptive compiler is coarser, as it does not generate code customized to individual values. Furthermore, the binaries that it generates contain all the possible variations of specialized code, whereas we only generate one specialized routine per function.

A technique that aims to explore runtime characteristics of a program is the hotspot optimization [Lau et al., 2006]. This technique, mostly used on virtual machines, explores the intuition that if a snippet is frequently executed or takes a considerable amount of time to run, it must be optimized. Thus, the hotspot optimization tries to employ aggressive transformations on regions of code where a high proportion of executed instructions occur or where most time is spent during the program's execution.

Trace scheduling [Fisher, 1981] and trace-based [Chang and Hwu, 1988] optimizations also specialize code based on dynamic aspects of the executed code. These optimizations determine, statically, sequence of instructions, including branches but not including loops, that must be executed sequentially. This way, it can optimize this so called *traces* with optimizations made for straight-line code sequence.

On Bolat and Li [2009], the authors propose a technique based on feedback that applies different optimizations according to the code behavior on distinct contexts. The technique depends on a previous analysis that determines sequences of basic blocks that may have different behaviors during multiple runs. According to the authors, if a snippet behaves the same between multiple runs, there is no need to optimize it in more than one way. Using the mentioned analysis, it is possible to apply different optimizations to a snippet, according to each context.

St-Amour et al. [2012] have designed a compiler that helps the developer to code in a more optimized way. The compiler suggests coding standards that can enable more aggressive optimizations. This tool, however, is not automatic, as the developer must explicitly agree with the modifications proposed by the compiler.

2.4 Context-aware optimizations

Compiler developers have long understood that procedure calls pose a barrier to code optimizations. Within a single procedure, control flow information can be relatively easy derived. Compilers are capable of, without great difficulty, determining variable lifetimes, constant expressions and common subexpressions. However, procedure calls must be treated as black boxes: anything could happen inside the call. Within each call site, the compiler suffers from a degradation in the quality of information it can

derive. This affects the optimizations it might be able to apply on the code.

Traditionally, two approaches have been used to break down the call site barrier and, thus, enable context-aware optimizations: *inlining* and *inter-procedural data-flow analysis*. *Inlining* replaces call sites with a copy of the body of the procedure. The code is then optimized in the context of the calling function. *Inter-procedural data-flow analysis* tries to determine a set of compile-time facts about the context where a function is called. This information is then made available to the intra-procedural optimizer.

Both techniques have limitations. Inlining can lead to code growth, increased compile time, and degradation in code quality [Cooper et al., 1991]. Optimizations based on inter-procedural data flow analysis have a drawback: they must take into account that each procedure can only be implemented once.

2.4.1 Function Cloning

An alternative to inlining and inter-procedural data-flow analysis in order to enable context-aware optimizations has been proposed by Mary Hall on her doctoral thesis [Hall, 1991, Cp.5] and latter discussed by Cooper et al. [1993]. The cloning technique improves on inter-procedural data-flow analysis by no longer assuming that each procedure should be implemented only once. This way, the compiler is free to create multiple specialized copies of a procedure and optimize each one separately. The calls to the original procedure can then be partitioned among all the specialized versions. The context of each call site is the responsible to determine which specialized version will be called.

Despite defining the concept of cloning, Mary Hall shows its application only on functions that receive constant as parameters. On a recent book, Grune *et al.* also describe cloning as a way to increase constant propagation reach [Grune et al., 2012, pg.325]. In fact, it is still very hard to find, on programming languages literature, discussions about the use of function cloning to implement context-aware optimizations. Industrial compilers, such as `icc` and `LLVM` do not use cloning. `Gcc` can clone a function that has only one call site and receives a constant as parameter. `Open64` is also able to clone functions, but just like `gcc`, just to help with constant propagation.

2.5 Alias analysis

Pointers are a feature of imperative programming languages. They are very useful, because they avoid the need to copy entire data-structures when passing information

from one program routine to another. Nevertheless, pointers make it very hard to reason about programs. The direct consequence of this difficulties is that compilers have a hard time trying to understand and modify imperative programs. Some compiler optimizations depend on knowing exactly which memory positions a pointer can reference: this is the case of some clone-based optimizations proposed in the work. To this end, compilers rely on alias analysis, also called pointer analysis or points-to analysis.

The goal of alias analysis is to determine which are the memory locations pointed by each pointer in the program. This analysis is usually described and solved as a constraint based analysis [Shivers, 1988].

Pointer analysis is often performed as a points-to analysis, which computes, for each pointer p and pointer dereference expression exp , the set of logical locations that may be accessed via p or exp . Two of the most well-known points-to analyses are described by Andersen [1994] and Steensgaard [1996]. These static points-to analyses compute an approximation of the set of objects to which a pointer may point. They are conservative in the sense that their results must be correct for any input and execution path of the program.

On this work, we have used an implementation of Andersen’s analysis in order to determine whether two pointers can alias. Furthermore, our implementation was improved with lazy cycle detection [Hardekopf and Lin, 2007], to make it possible to run in acceptable time even on large benchmarks, like the ones contained on SPEC CPU 2006.

2.6 Optimizations

Two of our optimizations, function fusion and pointer disambiguation are similar to techniques that have been discussed in previous work. However, none of these preceding works mention cloning. Our version of function fusion, seen in Section 3.4, is a step of *deforestation*, an optimization proposed by Philip Wadler in the context of functional programming [Wadler, 1988]. Deforestation has been very influential in the design of other optimizations for functional languages. In particular, we use the nomenclature introduced by Wei-Ngan Chin [Chin, 1992] in our work. The simple test that we use in Section 3.4 to ensure the safety of fusion is a design of our own. Researchers have also investigated the effectiveness of pointer disambiguation. Diego Huang, for instance, has proposed different ways to parallelize loops, once the arrays that these loops manipulate are shown not to be aliases [Huang, 2011]. Closer to our work, Markus Mock has studied the profitability of the restrict key work in the C programs available

in SPEC CPU 2000. By optimistically assigning the restrict modifier to every function argument, he has found that the expected performance gain lays between 1% and 8%. This result is similar to those that we present in Chapter 4.

Our clone-based optimizations belong into the family of *context-sensitive* code transformations. We enable our optimizations via static analysis which are classified as 0-CFA, following Shivers's terminology [Shivers, 1988]. This classification means that our contexts are formed by only one level of function calls. We distinguish two invocation sites of a function f within another function g , without considering which functions have invoked g . There are descriptions of more precise analyses in the literature. Usually they are time consuming [Whaley and Lam, 2004], or accept a certain amount of imprecision [Lhoták and Hendren, 2006]. Yet, we speculate, from earlier results obtained by Lhoták and Hendren [2006], that adding extra context levels to our analyses would not increase by much the number of functions that we can successfully specialize.

Chapter 3

Clone-Based Optimizations

In this chapter, we describe the clone-based optimizations that we have designed and tested. One of these optimizations - constant propagation - is not a contribution of this work; the others are novel. Our optimizations happen at the compiler level; hence, they are applied on intermediate representation programs. However, we will illustrate them using C code, because this notation is easier to understand.

3.1 This Work's Vocabulary

The *activation record* of a function call is the memory area that contains the data that the function needs to execute, such as local variables, parameters and return address. Usually, activation records are placed on a memory region named *the stack*. The *context* of a function invocation is the sequence of activation records piled onto the stack, at the moment the call was performed. A code optimization is said to be context dependent, or context aware, when it might yield different effects, depending on the context where it is performed. The cloning-based optimizations that we present in this work are context aware. It is possible, in principle, to consider the entire activation stack when determining the context of a function call [Whaley and Lam, 2004]. However, this approach is very expensive, because it burdens the compiler with a possibly exponential number of different contexts. Therefore, when determining the context in which a function f is called, we will be considering only the last function invoked before f , i.e., whose activation record is on the top of the calling stack. Hence, we say that our context has *depth one*.

We shall classify every function implementation present in the source code of the program that we are optimizing into one of two categories: *promising* or *indifferent*. A function is indifferent if it does not present any feature that we can optimize. Otherwise,

it is promising. For instance, if we are trying to remove unused return expressions, then a function that does not return any value is indifferent. Similarly, if we are trying to disambiguate pointers passed as parameters, then a function of zero arity, or a function that only receives scalars parameters, is indifferent.

We shall classify the calling sites of promising functions into two groups: *fruitful* or *fruitless*. Notice that, whereas our first classification - promising vs indifferent - refers to the implementation of functions, the second one - fruitful vs fruitless - refers to calling sites of functions. We say that the invocation site s , where a function f is called, is fruitful if s gives us the opportunity to replace f by an optimized clone f' . This opportunity is influenced by a suite of conditions, which can be determined statically, and that we shall name *enablers*. If s does not offer us any enabler, then this context is fruitless. If we find a fruitful invocation site, then we proceed to clone the callee. Henceforth we shall refer to the original function as *base*, and the new, derived code, as *clone*. The clone is subject to a number of transformations, which are due to the optimization that we apply. We shall use the generic term *effect* to refer to these transformations. We shall make these notions more clear in the next examples.

3.2 Clone-Based Constant Propagation

The literature describes constant propagation as the canonic example of a clone-based optimization [Grune et al., 2012; Hall, 1991; Kennedy and Allen, 2002]. This is also the only inter-procedural code transformation that relies on function cloning which we have found in industrial-strength compilers. Nevertheless, cloning is used in a very limited way. For instance, `gcc` clones a function if it has only one calling site in the entire program, and its linkage is internal. `Gcc`'s enabler is a calling site in which at least one of the parameters is a known constant. `Open64` does a similar trick to remove parameters from functions, if these parameters are always replaced by the same constants. In this work we revisit this technique, albeit on a more extensive way.

We shall use Figure 3.1 to illustrate three of the clone-based optimizations that we discuss in this section, including constant propagation. The program in Figure 3.1 contains implementations of functions `divMod` and `main`. This program also contains six calling sites, at lines 15, 16, 19, 20, 23 and 24. Three of these sites invoke the external function `printf`. We cannot optimize it, because we do not have access to its source code. In the other three sites we have calls to `divMod`. We can replace the fruitful calls with clones, whenever the right enabler occurs.

Following the well-established jargon, we use the term *formal parameter* to denote

```

1  int divMod(int a, int b, int* m) {
2  int quot = 0;
3  while (a > b) {
4    a -= b;
5    quot++;
6  }
7  *m = a;
8  return quot;
9  }

10 void main(int argc, char** argv) {
11 int mod;
12 int quot;
13 switch(argc) {
14 case 1:
15   quot = divMod(argv[0][0], 2, &mod);
16   printf("modulus = %d, quotient = %d\n", mod, quot);
17   break;
18 case 2:
19   quot = divMod(argv[0][0], argv[1][0], &mod);
20   printf("quotient = %d\n", quot);
21   break;
22 case 3:
23   divMod(argv[1][0], argv[2][0], &mod);
24   printf("modulus = %d\n", mod);
25   break;
26 }
27 }

```

Figure 3.1. A program that illustrates three different clone-based optimizations.

the names of parameters used in the declaration of a function. We use *actual parameter* to refer to the parameters with which the function is invoked. As an example, the function `divMod`, in Figure 3.1, has three formal parameters, `a`, `b` and `m`. At line 15, this function is invoked with three actual parameters: `argv[0][0]`, `2` and `&mod`. Our implementation of clone-based constant propagation is enabled at every call site in which we can prove that one or more actual parameters are constants known at compilation time. Continuing with our example, this optimization is enabled at line 15; hence, this is a fruitful call site.

This constant propagation generates a clone f' , from a base procedure f , which bears two effects. First, we replace every occurrence of the formal parameter p_f whose corresponding actual parameter p_a is a constant c by c itself on the body of f' . Second, we remove the declaration of p_f from the implementation of f' . Figure 3.2 illustrates these effects. We are using Wegman and Zadeck [1991]’s description of constant propagation, which lets us learn information from conditional tests. Constant propagation usually enables other optimizations. In this example, we show the final code that we obtain, after loop unrolling and instruction folding. Having produced a clone f' , we replace a call of f by an invocation of f' at every fruitful site. In our example, this replacement happened at line 15 of function `main`.

```

int divMod_clone_cp(int a, int* m) {
    int quot = 0;
    while (a > 2) {
        a -= 2;
        quot++;
    }
    *m = a;
    return quot;
}

11 void main(int argc, char** argv) {
    ...
15     case 1:
16         quot = divMod_clone_cp(argv[0][0], &mod);
    ...
28 }

```

The diagram illustrates the effects of clone-based constant propagation on the `divMod_clone_cp` function. It shows three stages of the code:

- (a) Code after constant propagation: The original code with a `while (a > 2)` loop. The loop body contains `a -= 2;`, `quot++;`, and `*m = a;`. The loop is annotated with (a).
- (b) Code after loop unrolling: The loop is unrolled into a single iteration. The code block contains `a -= 2;`, `quot++;`, `a -= 2;`, `quot++;`, and `*m = a;`. This block is annotated with (b).
- (c) Code after instruction folding: The unrolled code is simplified. The two `a -= 2;` statements are combined into `a -= 4;`, and the two `quot++;` statements are combined into `quot += 2;`. The code block contains `a -= 4;`, `quot += 2;`, and `*m = a;`. This block is annotated with (c).

Figure 3.2. The effects of clone-based constant propagation. (a) Code after constant propagation. (b) Code after loop unrolling. (c) Code after instruction folding.

3.3 Elimination of Unused Return Values

In some programming languages, such as Haskell or SML, every function returns a value. In others, such as C or Java, functions might, or might not, return values back to the caller. If a function does not return a value, then we shall refer to it as a *procedure*, otherwise, we shall name it a *valued function*. Procedures are only invoked for their side effects; however, side effects are not an exclusivity of them. Thus, a valued function can also be called for its side effect, while its return value is discarded by the invoker. We shall consider this situation an instance of a performance bug. Such performance bugs are rather common. For instance, the conspicuous `printf` function, present in C’s standard I/O library, returns the number of characters printed, or a negative value, in case an output error occurs. Yet, rarely do programmers use this value. The presence of unused return values is cited by Jin et al. [2012] as a consequence of programming malpractices.

We define the Elimination of Unused Return Values as an optimization that clones and improves valued functions whose returned data is not used in every calling context. This optimization regards as promising any valued function. The enabler of this transformation is standard dead code analysis. If a variable is defined in a program, but is not used in this program’s text, then we say that it is *dead*. Values used only in the definition of dead variables are dead as well. Calling sites that produce dead


```

void divMod_clone_urve(int a, int b, int* m) {
  int quot = 0;
  while (a > b) {
    a -= b;
    quot++;
  }
  *m = a;
}

void main(int argc, char** argv) {
  ...
  case 3:
  23   divMod_clone_urve(argv[1][0], argv[2][0], &mod);
  24   printf("modulus = %d\n", mod);
  25   ...
  28 }

```

Figure 3.3. The effects of the elimination of unused return values. (a) Code after elimination of the return statement. (b) Code after dead code elimination.

return values are considered fruitful to this optimization, otherwise they are fruitless. Given a promising function f , with at least one fruitful invocation site, the effect of our optimization is the following: we produce a clone f' that does not generate any return value, i.e., f' is a procedure. We then apply dead code elimination onto f' to remove from its body any computation that only contributes to the expression originally returned. Our dead code detection is made using classic analysis [Appel and Palsberg, 2002, p.417].

In Figure 3.1, we have a promising function, `divMod`, and an indifferent function, `main`. The latter is indifferent because it does not return a value. The call to `divMod` at line 23 is fruitful; the other invocations are fruitless, as they produce values which are not dead. The application of the elimination of unused return values in this example leads to the code shown in Figure 3.3. The primary effect of this optimization is the elimination of the return statement. Its secondary effects come out of dead code elimination. In this particular example, variable `quot`, declared in `divMod`, becomes dead, as it was only used in the return statement. A round of dead-code elimination produces the function seen in Figure 3.3 (b). After generating an optimized clone, we replace the original invocation of `divMod`, at line 23 of `main`, by the new function. Figure 3.3 shows all these effects.

An alternative optimization is inline expansion, which allows each context to be optimized alone. Figure 3.4 show inlining applied to the same `divMod` function. On

```

int main(int argc, char** argv) {
    int modulus;
    int quotient;
    switch(argc) {
        case 1: {
            int quot = 0;
            int a = argv[0][0];
            int b = argv[0][1];
            int* m = &modulus;
            while (a > b) {
                a -= b;
                quot++;
            }
            *m = a;
            printf("q = %d, m = %d\n", quot, *m);
            break;
        }
        case 2: {
            int quot = -0;
            int a = argv[0][0];
            int b = argv[1][0];
            int* m = &modulus;
            while (a > b) {
                a -= b;
                quot++;
            }
            *m = a;
            printf("modulus = %d\n", *m);
            break;
        }
        case 3: {
            int quot = -0;
            int a = argv[1][0];
            int b = argv[2][0];
            int* m = &modulus;
            while (a > b) {
                a -= b;
                quot++;
            }
            *m = a;
            printf("modulus = %d\n", *m);
            break;
        }
    }
}

```

```

int main(int argc, char** argv) {
    int modulus;
    int quotient;
    switch(argc) {
        case 1: {
            int quot = 0;
            int a = argv[0][0];
            int b = argv[0][1];
            int* m = &modulus;
            while (a > b) {
                a -= b;
                quot++;
            }
            *m = a;
            printf("q = %d, m = %d\n", quot, *m);
            break;
        }
        case 2: {
            -----
            int a = argv[0][0];
            int b = argv[1][0];
            int* m = &modulus;
            while (a > b) {
                a -= b;
            }
            *m = a;
            printf("modulus = %d\n", *m);
            break;
        }
        case 3: {
            -----
            int a = argv[1][0];
            int b = argv[2][0];
            int* m = &modulus;
            while (a > b) {
                a -= b;
            }
            *m = a;
            printf("modulus = %d\n", *m);
            break;
        }
    }
}

```

Figure 3.4. Program on figure 3.1, after inline expansion and dead code elimination.

the left side of the figure, we show the main function, originally shown on figure 3.1, after inline expansion. Each call site of the `divMod` function is replaced by its body. One of the advantages of inlining is that it allows the compiler to optimize each context in a different way, since there are no dependencies between contexts. On the example shown, the compiler was able to remove the computation of the `quot` variable in both contexts they are not used. However, as a result of inlining, we have a huge increase in code size on the optimized program. This increase can compromise the program not

only in size, but also in execution time, as it can damage the locality of reference on instructions cache [Hennessy and Patterson, 2003]. Furthermore, there are situations where reduced code size is still a big advantage, because it will be run on devices with low memory resources, like embedded ones.

3.4 Function Fusion

It is usual that programmers chain functions together, feeding one of them with the result produced by the other. Figure 3.5 illustrates an example of such pattern. In the figure, the outcome of `strlen` is immediately forwarded to `sumArray`. Following Chin's nomenclature [Chin, 1992], we call `strlen` a *producer*, and `sumArray` a *consumer* function. If this situation happens often enough, we can decide to merge `strlen` and `sumArray`. We call this optimization *function fusion*. In this work, we chose to enable the fusion of a producer f and a consumer g when the following conditions apply:

- (i) f produces, as return value, a variable v , e.g., $v = f(\dots)$;
- (ii) v is only used as a parameter of g , e.g., $g(\dots, v, \dots)$.

Notice that condition (ii) is necessary to avoid eliminating the definition of a value that has further uses, besides being a parameter of g .

```

1  int sumArray(char* v, int n) {
2  int i;
3  int sum = 0;
4  for (i = 0; i < n; i++) {
5    sum += v[i];
6  }
7  return sum;
8  }

9  int strlen(char* a) {
10 int i = 0;
11 while (a[i] != '\0') {
12   i++;
13 }
14 return i;
15 }

16 int main(int argc, char** argv) {
17   if (argc == 2)
18     printf("%d\n", sumArray(argv[1], strlen(argv[1]]));
19   else
20     printf("%d\n", sumArray(argv[1], 0));
21 }
```

Figure 3.5. Example that benefits from function fusion.

```

1  int sumArray_clone(char* v, char* a) {
2  int i;
3  int sum = 0;
4  int n = strlen(a);
5  for (i = 0; i < n; i++) {
6    sum += v[i];
7  }
8  return sum;
9  }
                                     (a)

1  int sumArray_clone_ff(char* v) {
2  int i;
3  int sum = 0;
4  int n = 0;
5  while (v[n] != '\0') {
6    n++;
7  }
8  for (i = 0; i < n; i++) {
9    sum += v[i];
10 }
11 return sum;
12 }
                                     (c)

1  int sumArray_clone_ff(char* v, char* a) {
2  int i;
3  int sum = 0;
4  int n = 0;
5  while (a[n] != '\0') {
6    n++;
7  }
8  for (i = 0; i < n; i++) {
9    sum += v[i];
10 }
11 return sum;
12 }
                                     (b)

16 int main(int argc, char** argv) {
17 if (argc == 2)
18   printf("%d\n", sumArray_clone_ff(argv[1]));
19 else
20   printf("%d\n", sumArray(argv[1], 0));
21 }
                                     (d)

```

Figure 3.6. (a) Example of Figure 3.5, after preliminary merging. (b) Inline expansion. (c) Elimination of redundant parameters. (d) Replacement of clones in call sites.

Once we find an enabler that lets us fuse f into the body of g , our function fusion produces the following effects, where P_f is the set of formal parameters of f , and P_g is the set of formal parameters of g :

1. create a function `g_clone`. We let the parameters of `g_clone` be formed by $P_f \cup P_g$.
2. insert a call of f as the first statement of `g_clone`. We let this call to be named the *surrogate*;
3. perform inline expansion of the surrogate within `g_clone`;
4. whenever applicable, we eliminate redundant parameters.

This last step is an optional optimization: if all the fruitful calls pass common parameters to consumers and producers, then the clone will have redundant parameters, which can be factored out. Figure 3.6 illustrates this transformation. In this example,

the producer and the consumer have common parameters, which have been factored out in Figure 3.6(c).

3.5 Pointer Disambiguation

The natural evolution of programming languages might lead to the occurrence of performance bugs in legacy code. In this case, the performance bug happens backwardly on time, because a program, developed in an early version of a language, might not benefit from new features added onto this language thereafter. An example of this situation is due to the addition of the `restrict` keyword to C, as of the C99 standard [ISO, 1999, pp. 110]. This type modifier can be used in the declaration of parameters of pointer type. It gives programmers the opportunity to state that a pointer has no aliases. Such information promotes code optimizations, as the compiler no longer needs to consider the effects of aliasing on code transformations regarding that pointer. This way, it is free to perform more aggressive optimization on the code due to memory disambiguation. Previous work found that, in some cases, memory disambiguation can result in significant speedups for kernel array codes [Bernstein et al., 1994]. Prior to the C99 standard, however, developers did not have access to this keyword. Hence, there exists a substantial body of binary code in deployed systems that could have been further optimized had it been written today. The optimization that we discuss is an attempt to deal with this problem.

To motivate this optimization, consider the code on Figure 3.7. Procedure `vmul` takes two arrays of floating point numbers, `a` and `b`, and their size `n` as inputs and computes the square of each element of `a`, storing them in array `b`. Without further knowledge and without special hardware support, the compiler must assume that `a` and `b` may refer to the same array or overlapping arrays, so that the loop cannot be parallelized or software-pipelined. The compiler has to ensure that an update of position `b[i]` is performed before the next value `a[i]` is loaded. This constraint also prevents the compiler from generating loads for multiple array elements of `a` at the same time, since the subsequent store to `b[i]` may modify elements of array `a`.

```
1 void vmul(int n, double* a, double* b) {
2     int i;
3     for (i = 0; i < n; i++)
4         b[i] = a[i] * a[i];
5 }
```

Figure 3.7. A simple vector multiplication routine.

```

1 void vmul(int n, double* restrict a, double* restrict b) {
2     int i;
3     for (i = 0; i < n; i++)
4         b[i] = a[i] * a[i];
5 }

```

Figure 3.8. The same vector multiplication routine, with its arguments modified with the `restrict` keyword, meaning that they never alias.

The definition of the `restrict` keyword, according to Oracle [2013] specifies that “an object that is accessed through a `restrict` qualified pointer requires that all accesses to that object use, directly or indirectly, the value of that particular `restrict` qualified pointer. Any access to the object through any other means may result in undefined behavior. The intended use of the `restrict` qualifier is to allow the compiler to make assumptions that promote optimizations”.

Figure 3.8 shows an example of how `restrict` is supposed to be used. The pointer-typed arguments `a` and `b` are modified with the `restrict` keyword. This qualification tells the compiler that the array pointed to by `a` is only accessed via pointer `a` or any pointers derived from it. Similarly, the array pointed to by `b` is only accessed via pointer `b` or pointers derived from it. Using these assertions, the compiler is able to state that `a` and `b` may not overlap. Consequently, it can perform parallelization and other transformations whose correctness depends on `a` and `b` not being aliased. Compared to the routine shown in Figure 3.7, the `restrict`-qualified routine in Figure 3.8 executes 24% faster on a Sparc workstation and even 2.9 times faster on an Itanium-based workstation with two processors, according to Mock [2004].

In order to achieve the benefits of memory disambiguation, we have implemented an optimization that we call *pointer disambiguation*. Its goal is to inform the compiler that formal parameters of pointer type do not alias each other. To this end, LLVM, our baseline compiler, provides the `noalias` argument attribute, which is defined by its intermediate program representation. Quoting LLVM’s programming manual¹, “the definition of `noalias` is intentionally similar to the definition of `restrict` in C99 for function arguments, although it is slightly weaker”. In the context of this optimization, a function is promising if it contains at least two formal parameters of pointer type. The optimization is enabled on a function call $f(a_1, \dots, a_n)$, where $a_i, 1 \leq i \leq n$ are actual parameters, if there are no a_i and a_j of pointer type that alias each other, or any global variable. Calling sites where this condition applies are considered fruitful. This optimization has the following effect on a function $f(b_1, \dots, b_n)$, where each $b_i, 1 \leq i \leq$

¹Available on-line at <http://llvm.org/docs/LangRef.html>

```

1 void copy(char* a, char* b, char* r, int N) {
2     int i;
3     for (i = 0; i < N; i++) {
4         r[i] = a[i];
5         if (!b[i]) {
6             r[i] = b[i];
7         }
8     }
9 }
10
11 int main(int argc, char** argv) {
12     char* buf = (char*) malloc(SIZE*sizeof(char));
13     if (argc < 2) {
14         strcpy(buf, argv[0]);
15         copy(argv[0], buf, buf, SIZE);
16     } else {
17         copy(argv[0], argv[1], buf, SIZE);
18     }
19     print(buf, SIZE);
20 }

```

Figure 3.9. Example in which the potential aliasing between pointers can prevent code optimizations.

n is a formal parameter: it creates a clone $f'(b'_1, \dots, b'_n)$ such that $b'_i = b_i$ if b_i is not a pointer, and $b'_i = \text{noalias } b_i$ otherwise. The compiler is then free to optimize f' as much as the new knowledge allows it.

In order to determine if two pointer-typed arguments can alias each other, we have implemented and used an alias analysis algorithm like the one described on section 2.5.

Figure 3.9 illustrates our pointer disambiguation. This program has been adapted from a recent work due to Chabbi and Crummey, who have used it as an example of performance bug [Chabbi and Mellor-Crummey, 2012]. Function `copy` might write the same position of array `r` twice, depending on the result of the branch at line 5. The first write happens in line 4, and the second might happen in line 6. This redundancy is necessary, because the arrays `b` and `r` can be aliases. If they are, indeed, aliases, then the store at line 4 may change the outcome of the test at line 5. This happens, for instance, in the invocation of `copy` at line 15 of Figure 3.9. In that call site, the formal parameters `b` and `r` are bound to the same array `buf`. On the other hand, if the compiler had the knowledge that such aliasing is not possible, as in line 17, then it could apply very aggressive optimizations onto the code of `copy`.

The call at line 17 of Figure 3.9 meets the enabling conditions of our pointer

```

1 void copy_clone_pd(char* restrict a,
   char* b, char* restrict r, int N) {
2   int i;
3   for (i = 0; i < N; i++) {
4     int tmp = a[i];
5     if (!b[i]) {
6       tmp = b[i];
7     }
8     r[i] = tmp;
9   }
10 }

```

```

11 int main(int argc, char** argv) {
   ...
17  copy_clone_pd(argv[0],
   argv[1], buf, SIZE);
   ...
20 }

```

Figure 3.10. Optimized version of `copy`, the function seen in Figure 3.9.

disambiguation. Therefore, we can clone `copy`, and mark its parameters with the `noalias` modifier². Figure 3.10 shows the result of this optimization. The two stores onto `r` have been replaced by updates on the local variable `tmp`, at lines 4 and 6. At line 8 the value in `tmp` is transferred to `r`. Even though this optimization consists on small, and very local changes in the code of `copy`, it delivers very good results. To emphasize this last point, we have compared `copy` and `copy_clone_pd` on an Intel Core 2 Duo, 2.27GHz, having `N` equal to 100,000. In this setting, the cloned function is almost 30% faster than its original version.

3.6 Elimination of Dead Stores

Functions usually produce side-effects by writing values in memory. Such updates are used, for instance, to indicate exceptional conditions in C, or to simulate functions that return multiple values. For example, in Figure 3.1, function `divMod` returns the remainder of the division of `a` by `b` in its third parameter `m`. We consider that a performance bug occurs whenever a function `f` updates a memory location that is *dead* outside the scope of `f`. A memory location is dead if it is not read before being updated again. Although it is difficult to pinpoint dead memory locations statically, Chabbi and Mellor-Crummey have shown, via a dynamic analysis of execution traces, that this problem is widespread among binaries produced out of C and C++ programs [Chabbi and Mellor-Crummey, 2012].

²Because we are using C in our examples, and not LLVM assembly, we use `restrict` instead of `noalias` in Figure 3.10


```

int divMod_clone_dse(int a, int b) {
  int quot = 0;
  while (a > b) {
    a -= b;
    quot++;
  }
  return quot
}

void main(int argc, char** argv) {
  ...
19  case 2:
20  quot = divMod_clone_dse(argv[1][0], argv[2][0]);
21  printf("quotient = %d\n", quot);
  ...
28 }

```

Figure 3.11. The effects of the elimination of dead stores.

In this work, we define the elimination of dead stores as a clone-based optimization that removes code to update dead *external memory* from the body of functions. Any function f that updates external memory is considered promising. External memory can be allocated inside the scope of f , or outside it. Any memory that has been allocated outside the scope of f is external to f . Any memory that f allocates in the heap is external, as long as this location *escapes* f . A memory location m escapes the scope of a function f if f returns a pointer to it, or assigns its address to a parameter passed via a pointer [Blanchet, 1998]. A call site is fruitful if it invokes a promising function f , which updates external memory m , and at least one of the following enabling situations occurs:

- Location m is declared locally in the scope of f 's caller, and is not read after f returns;
- Location m is overwritten in the scope of f 's caller, after f returns, without being read before.

We have limited our analysis to the local scope of f 's caller, to simplify it. This limitation means that we do not discover enablers via a global dead memory analysis; instead, we use a local analysis within f 's caller. We use an Andersen style [Andersen, 1994] - global - pointer analysis to avoid false negatives. In other words, we assume that location m is updated if any of its aliases is written. When applied on a function f , this optimization produces a clone f' with the following effects: any update of dead

external location m is removed from f' , and dead code elimination is applied to get rid of subsequent expressions that is made useless.

Continuing with our example, function `divMod` in Figure 3.1 is promising, as it updates location `m`, which is external to its scope. Function `main` is indifferent, as it does not update any external memory location. The call of `divMod` at line 20 is fruitful, because location `mod` is not read after the function returns. The other calling sites are fruitless. Figure 3.11 shows the effect of applying dead store elimination on the program seen in Figure 3.1. In this case, the only profit of our optimization was the removal of the assignment `*m = a` in line 7 of Figure 3.1, and the replacement of the call of `divMod` at line 20 by a new call of the clone `divMod_clone_dse`.

3.7 Estimating the Profit of a Clone

We need a way to estimate the profitability of a clone. Such metric is useful for two reasons. Firstly, some clones offer too little benefit over the original function, and we believe that they are not worth keeping. Our metric helps us to remove unwanted clones; hence, it avoids excessive code expansion. Secondly, this number gives the compiler a more concrete way to point back to the program developer potential performance bugs. Very profitable clones offer to the software engineer strong indication that code is being reused in improper ways. In this work, we chose to use Wu and Larus static profiler [Wu and Larus, 1994] to measure the benefit of a clone.

Wu and Larus's static profiler assigns to each edge in the control flow graph of a program a *frequency*. This frequency is an estimate of how many times that edge will be traversed during the execution of the program. In order to determine this frequency, Wu and Larus resort to heuristics. These rules gauge the chance that branches will be taken. Some heuristics are based on the structure of the control flow graph of the program, and others are based on the type and/or value of variables used in conditional tests. For instance, Wu and Larus assume that a branch that goes back to a loop header, or that leaves the loop will stay in the loop 88% of the time. They also predict that a comparison of a pointer against null is likely to fail with 60% of chance. In total, Wu and Larus have defined nine different approximation rules. It is possible that different rules apply to the same branch. In this case, the authors have a way to combine these different probabilities, a feat that they accomplish with Dempster-Shafer theory of evidences [Shafer, 1976].

Figure 3.12 applies the static profiler on function `divMod`, seen in Figure 3.1. Part (a) of the figure shows the probabilities that each edge in the CFG is taken,

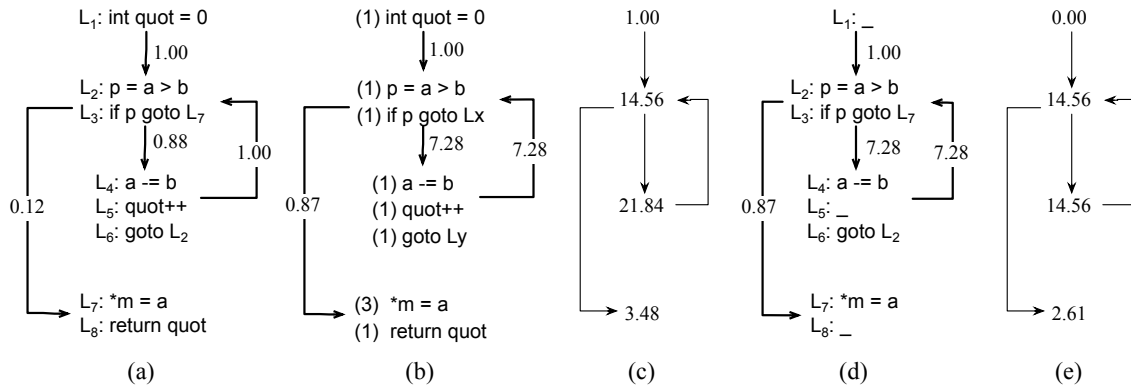


Figure 3.12. (a) The control flow graph of function `divMod`, originally seen in Figure 3.1, augmented with probability of edge being taken. (b) Cost of each instruction in the CFG, and edge frequencies. (c) Total cost of function `divMod`. (d) CFG of `divMod_clone_urve`, seen in Figure 3.3. (e) Total cost of function `divMod_clone_urve`.

and part (b) shows the edge frequencies. The calculation of edge frequencies is non-trivial, because it depends on the notion of *block frequency*. The definition of these two quantities is mutually-recursive. The frequency of which block u is visited is the sum of the edge frequencies of every edge that reaches u . The frequency of an edge $u \leftarrow v$, that leaves block u towards block v is the frequency of u multiplied by the probability that $u \leftarrow v$ is taken. Wu and Larus give a technique to compute the least fixed point of the equations that determine edge and block frequency.

The result of this technique, when applied on our example, lets us to compute the *cost* of each basic block, as we show in Figure 3.12(c). We chose to approximate the cost of a basic block b by multiplying its execution frequency by the individual cost of every instruction within b . We estimate the cost of instructions by reading the architecture manual. In our example, we assigned a 1-cycle cost to every instruction, but stores, which are worth 3-cycles.

Figure 3.12(d) shows the edge frequency of the cloned function `divMod_clone_urve`. This clone is produced from `divMod` by the application of Elimination of Unused Return Values, the optimization that we discussed in Section 3.3. Figure 3.12(e) shows the cost of the optimized clone. As we can see, we predict that `divMod_clone_urve` will be slightly faster than `divMod`. We can even guess the speedup that the clone delivers, when compared to the original function: $(1.00 + 14.56 + 21.84 + 3.48)/(14.56 + 14.56 + 2.61) = 28.83\%$.

3.7.1 Validating the static profiler

We have made some experiments in order to validate the reliability of Wu and Larus’s static profiler. In order to do that, we have compared branch probabilities generated by the static profiler with real branch frequencies obtained using a dynamic profiler. We have made our tests against the benchmarks present on SPEC CINT 2006 with the provided inputs.

The dynamic profiler gives us the number of times a branch is taken (*branch_frequency*) and the number of times the condition of a branch is analyzed (*condition_frequency*). Based on these two numbers, we got a *dynamic_probability*, given by the expression $branch_frequency/condition_frequency$. Wu and Larus’s static profiler gives us the static probability a branch should be taken. Given the static and dynamic probabilities, we have calculated the error of the static profiler with the formula $|dynamic_probability - static_probability|$.

We show our results on Figure 3.13. We have analyzed the error of the static profiler with and without considering untaken branches. On the former case, we consider every branch of the programs in order to calculate the error. On the latter case, we only calculate the error of branches taken during execution.

Ignoring untaken branches, Wu and Larus’s static profiler had an average error of 25,34%. When considering untaken branches, however, the average error was much higher, 46,46%. This was already expected, as the static probability of untaken

Bench	Error with untaken branches	Error without untaken branches
perlbench	29,41%	54,64%
bzip2	28,06%	41,17%
gcc	30%	49,08%
mcf	22,01%	32,64%
gobmk	27,03%	45,58%
hmmer	24,13%	56,08%
sjeng	27,86%	46,69%
libquantum	27,86%	48,87%
h264ref	22,72%	52,06%
omnetpp	21,7%	42,5%
astar	18,2%	31,79%
xalancbmk	31,48%	56,45%

Figure 3.13. Errors of Wu and Larus’s static profiler on SPEC CINT 2006.

branches is considered an error as a whole. We have made the same experiments with LLVM's static profiler and found similar numbers: 24,82% of average error without considering untaken branches and 47,14% of average error considering branches not taken during execution. This way, we believe Wu and Larus's static profiler provides enough reliability to let us calculate the static costs of our optimized clones.

Chapter 4

Experiments

The goal of this chapter is to show the applicability and the effectiveness of the proposed optimizations. We have implemented them as modules of the LLVM compiler, version 3.3. This way, they are easily loaded and used during the optimization step, using the `opt` tool. All the experiments reported in this work have been performed on an Intel Xeon CPU e5-2665, with 132GB of RAM, running at 2.4GHz. The operating system used was Ubuntu 12.04. We have run our optimizations against the well-known benchmarks in SPEC CPU 2006 [Henning, 2006] and LLVM Test-Suite¹. This last set of benchmarks is commonly used to test new features and optimizations implemented on top of LLVM.

4.1 The Cloning Pipeline

Figure 4.1 shows our *code transformation pipeline*, i.e., the sequence of steps that we perform to apply one clone-based optimization. All our clone-based transformations are completely independent, and they can be applied together, or separately. In order to apply our optimizations, we have split the required steps into two groups, **optimization steps** and **additional steps**.

The *optimization steps* are responsible for creating the cloned functions. They are described as follows:

1. **Identification**: we run some specific static analysis to find promising functions, and fruitful call sites.
2. **Cloning**: each promising function is cloned.

¹Documented on-line at <http://llvm.org/docs/TestingGuide.html>

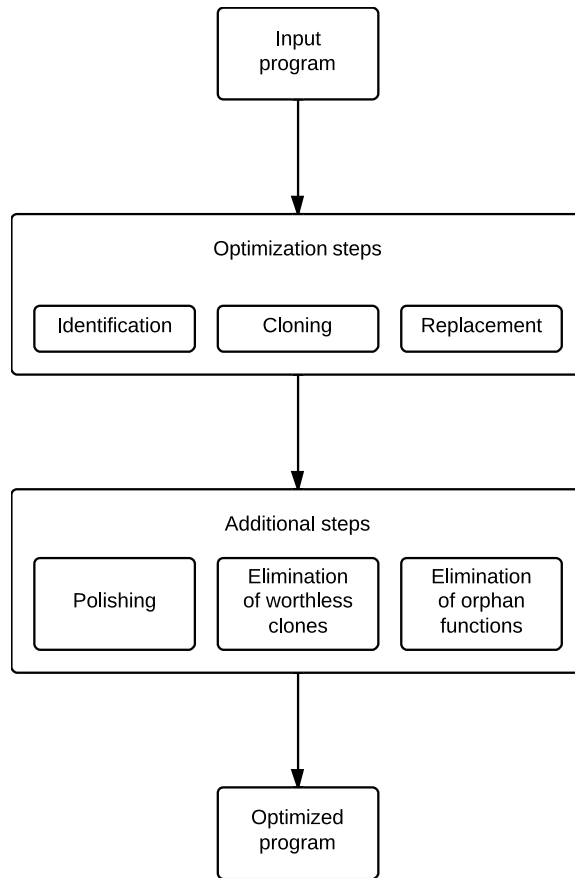


Figure 4.1. Transformation pipeline that we use to apply function cloning.

3. **Replacement:** fruitful calls are replaced by invocations of the clone.

Once we finish these steps, we perform some *additional steps* in order to improve our generated code. They are described as follows:

1. **Polishing:** we run the optimizations available in LLVM -O2 over the entire program. It is this set of optimizations that will polish the code of the clones. For instance, when we apply unused return values elimination, we remove return instructions from the clone, and set its type to `void`; however, we do not eliminate subsequent dead instructions. This last stage is performed by the optimization steps described above.
2. **Elimination of worthless clones:** After this optimized code is generated, we run a final pass over the program, estimating the costs of all cloned and base functions, like we described on section 3.7. This estimation gives us a metric to remove clones that we think are not worth keeping. For instance, given a

base function and its corresponding clone, we remove the cloned version if its static cost is greater than the cost from the original function. This difference can happen because both the original and cloned versions went through the polishing step. This way, it's possible that our clone-based optimization may have inhibited the applicability of some other optimizations, in a way that the final code for the base function is the more optimized version. If that's true, we remove the cloned version and restore fruitful calls to invocations of the base function. Sometimes, it is not possible to restore the original calls, as the base and cloned versions of the function have different arities.

3. **Elimination of orphan function:** Before producing an executable code, we remove *orphan functions*. If all the calling sites of a given promising function are fruitful, then the original function will no longer be reachable. In this case, we say that this function is orphan.

During the *polishing* step, we may choose to remove the inline expansion optimization from LLVM -O2. We may do this because inlining makes it hard to probe the effectiveness of our transformations, as it removes from the program several calls to original and cloned functions. On the rest of this chapter, if we have chosen to remove inline expansion, we will explicitly say we have done so.

4.2 The Applicability of our Optimizations

In this section, we provide numbers to show the applicability of the proposed optimizations. We have tested the clone-based optimizations on the well-known SPEC CPU 2006 benchmark and the benchmarks contained in LLVM Test-Suite. The latter is a collection of benchmarks divided in two types of tests: single source programs and multiple source ones. The *SingleSource* benchmarks are known to be simple chunks of code contained on a single file. The *MultiSource* benchmarks contain larger benchmarks and whole applications. We have divided our numbers according to the benchmarks used to generate them.

4.2.1 Test-Suite SingleSource

The Test-Suite *SingleSource* is a collection of 311 simple programs. The whole collection has 47,845 lines of code and 11,208 functions. Thus, the average number of functions in a single benchmark is 36. Figures 4.2–4.5 provide applicability data for each of our proposed optimizations when applied to these benchmarks. We do not show numbers

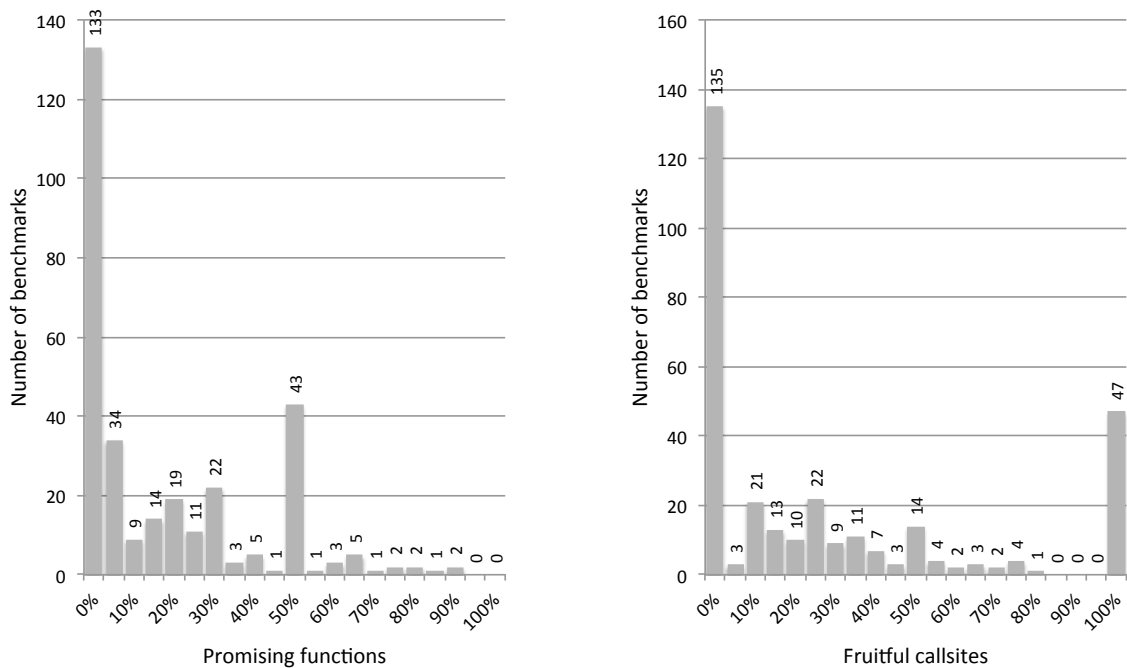


Figure 4.2. Constant Propagation: Applicability on Test-Suite SingleSource.

for elimination of dead stores, as it could not find any optimization opportunity on the collection.

We have chosen to show the data in form of histogram graphics. For each benchmark, we have calculated the percentage of promising functions, i.e., the ones that can benefit from a given optimization. This number is given by the equation $Clone/Func$, where $Clone$ is the number of functions cloned while $Func$ is the total number of functions contained in the program. We also wanted to show the applicability of our optimizations in terms of fruitful call sites, i.e., call sites that can benefit from the optimized clones. So, we have designed histogram graphics showing the percentage of fruitful call sites. This number is given by the equation $F-ful/Calls$, where $F-ful$ is the number of fruitful call sites, while $Calls$ is the total number of calls of the benchmark. The total number of function calls in the entire collection is 22,522, with an average of 72 function calls in each benchmark.

In the histograms shown, the Y axis is the number of benchmarks, while the X axis is the percentage of promising functions or fruitful call sites. We just show the percentages in multiples of 5, in order to make our graphics smaller and easier to understand. Thus, in Figure 4.2 left, we can read that 133 benchmarks (out of 311) had 0%–5% of functions that could benefit from clone-based constant propagation. As another example, Figure 4.2 right tells us that 47 benchmarks have had all their

function calls replaced by invocations of clones optimized by constant propagation.

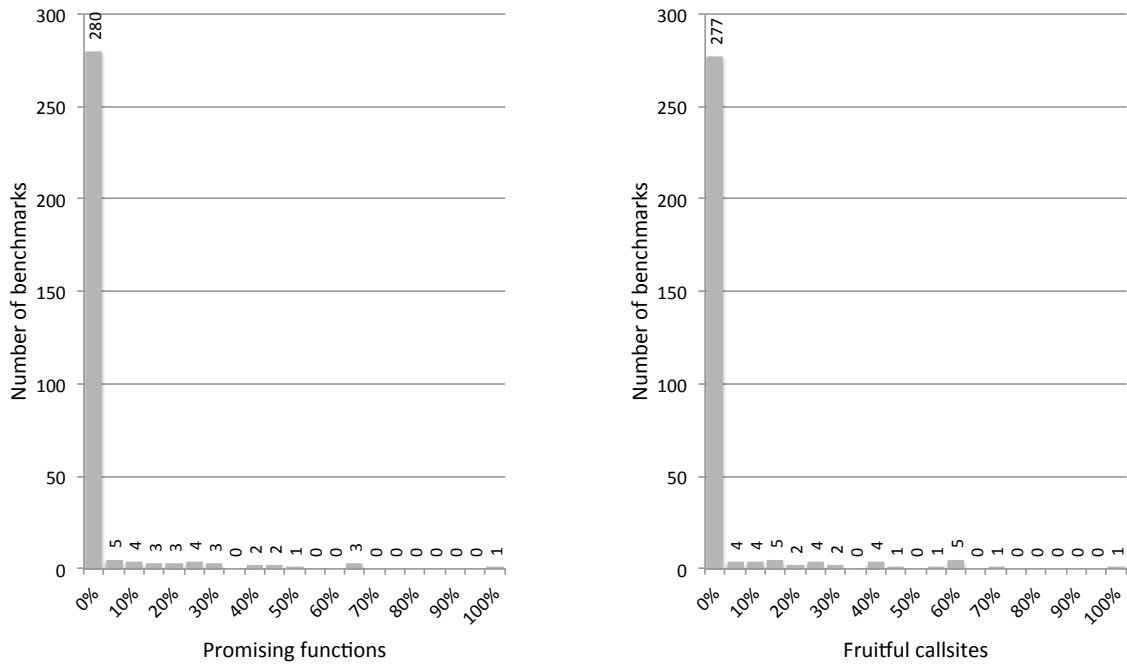


Figure 4.5. Function Fusion: Applicability on Test-Suite SingleSource.

From the figures 4.2–4.5, we can see that constant propagation has a high applicability, while all the other optimizations see less reach. The average percentage of promising functions are: constant propagation 17.65%, elimination of unused retvals 4.43%, pointer disambiguation 10.45% and function fusion 2.81%. Our optimizations have a relevant applicability in terms of call sites. For instance, the average percentage of call sites touched by our optimizations are: constant propagation 27.95%, elimination of unused retvals 6.62%, pointer disambiguation 11.88% and function fusion 3.42%.

4.2.2 Test-Suite MultiSource

The Test-Suite *MultiSource* is a collection of 182 programs. These programs consist of large benchmarks and whole applications. The whole collection has 1,343,150 lines of code and 46,034 functions. Thus, the average number of functions in a single benchmark is 252. The total number of function calls in the entire collection is 188,346, with an average of 1,034 function calls in each benchmark.

Figures 4.6–4.10 provide applicability data for each of our proposed optimizations when applied to these benchmarks. We show histogram graphics for the percentage of

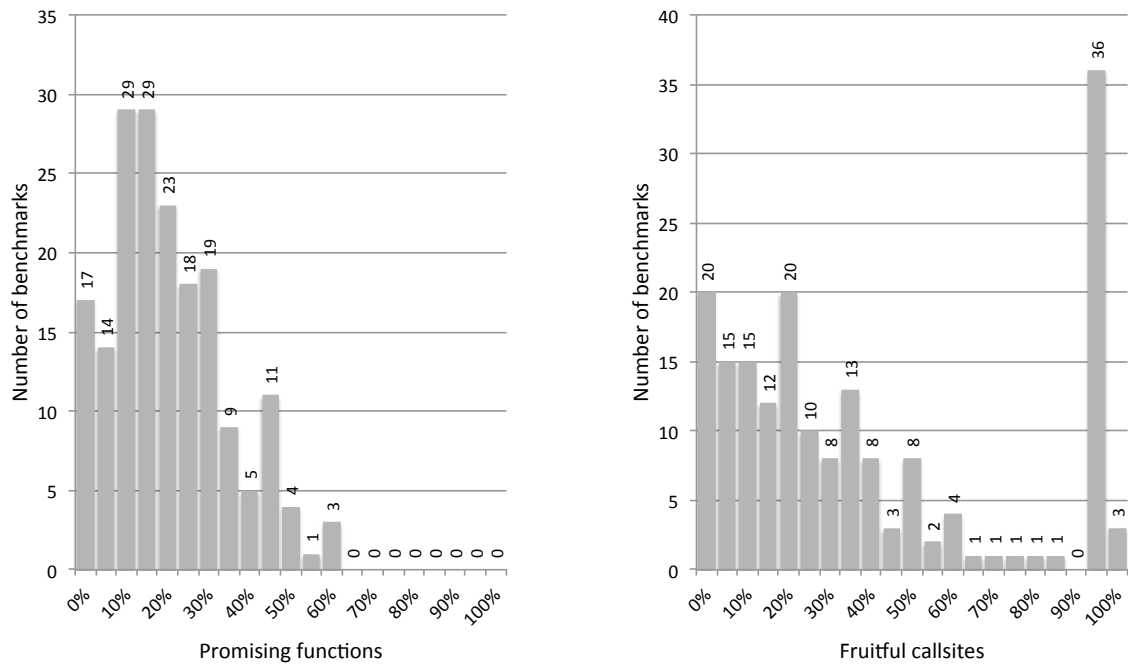


Figure 4.6. Constant Propagation: Applicability on Test-Suite MultiSource.

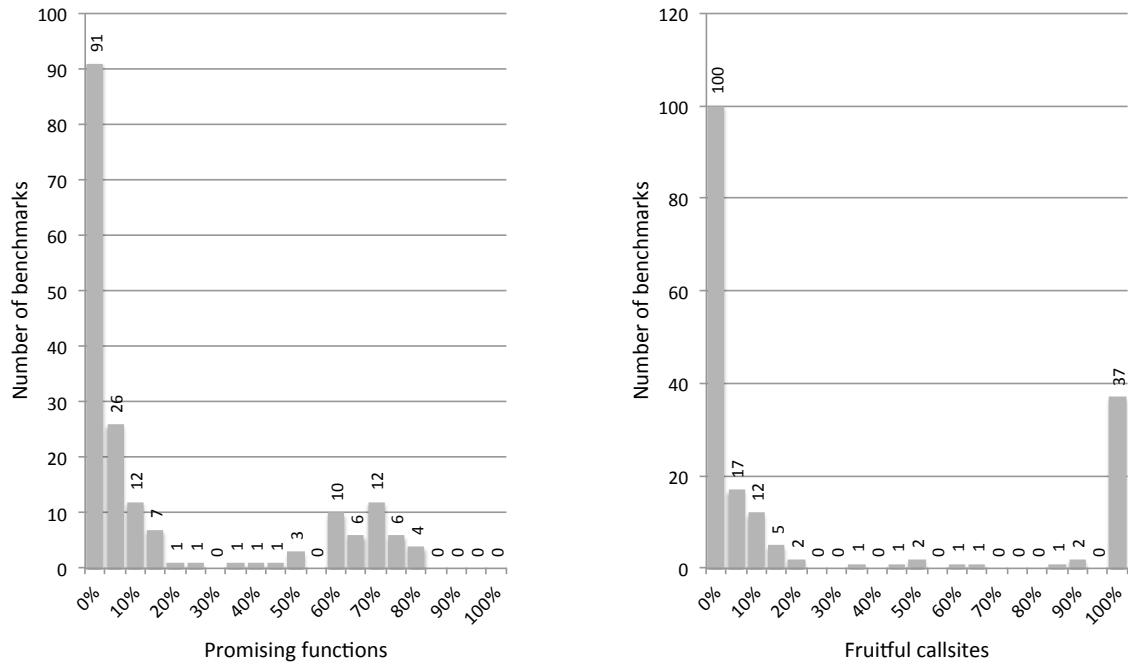


Figure 4.7. Elimination of unused retvals: Applicability on Test-Suite MultiSource.

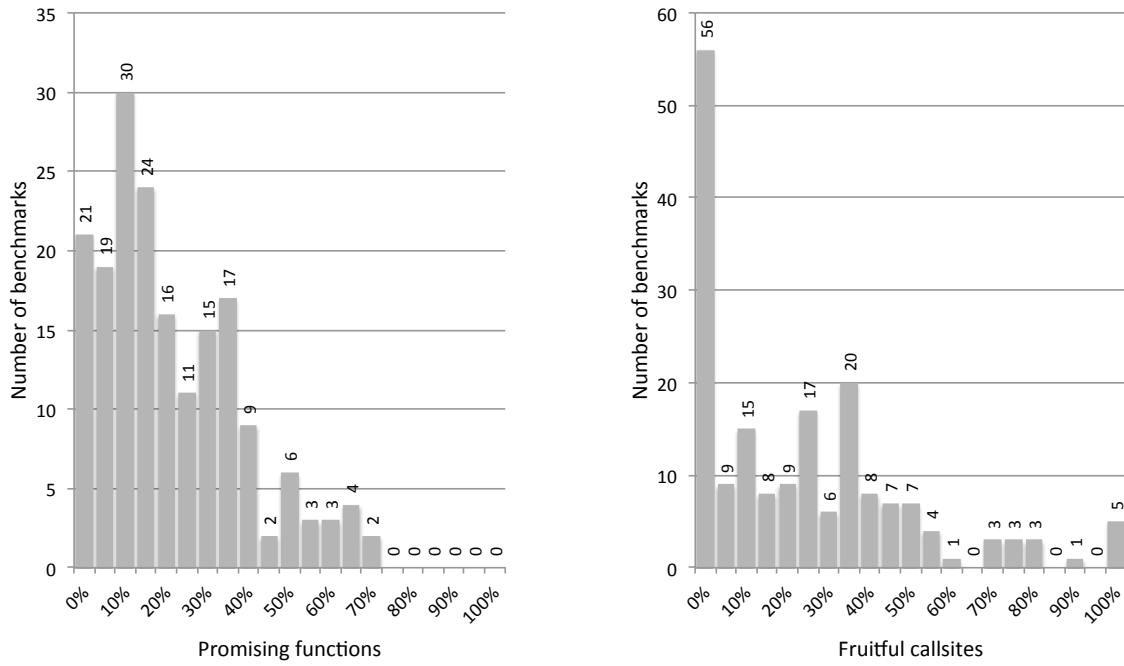


Figure 4.8. Pointer Disambiguation: Applicability on Test-Suite MultiSource.

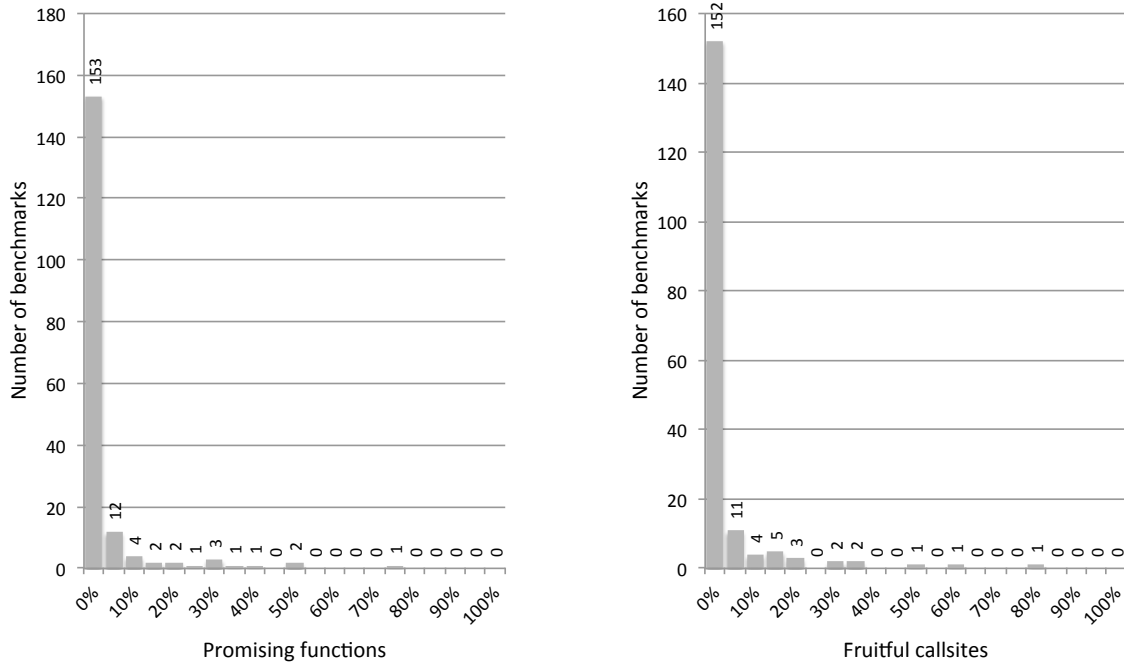


Figure 4.9. Function Fusion: Applicability on Test-Suite MultiSource.

promising function and fruitful call sites, just like we have done on Section 4.2.1.

From the figures 4.6–4.10, we can see that the proposed optimizations have much more reach when dealing with larger benchmarks. All the optimizations obtained a high applicability, with averages of promising functions being 20.43% for constant propagation, 17.96% for elimination of unused retvals 21.37% for pointer disambiguation, 2.93% for function fusion and 0.65% for elimination of dead stores. We could see a high applicability in terms of call sites. Considering all function calls in the collection of benchmarks, the proportions of call sites touched by our optimizations were: constant propagation 38.73%, elimination of unused retvals 25.21%, pointer disambiguation 23.79%, function fusion 3.02% and elimination of dead stores 0.38%.

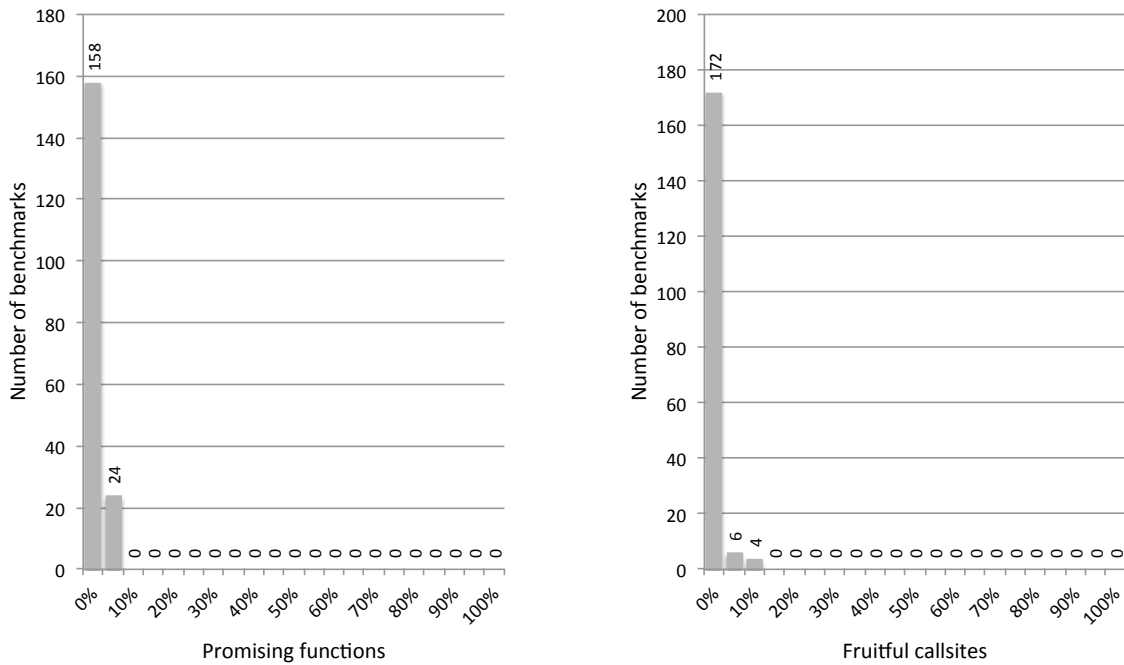


Figure 4.10. Elimination of Dead Stores: Applicability on Test-Suite MultiSource.

4.2.3 SPEC CPU 2006

The *SPEC CPU 2006* is a well-known collection of programs. These programs are often used, both on industrial and academic community, in order to test the performance of either hardware or software. The whole collection contains 1,910,549 lines of code and 75,052 functions. The benchmarks in SPEC are divided in two classes of programs: *SPECint* and *SPECfp*. The former is used for testing integer performance, whereas the latter tests floating-point performance.

Bench	Func	Clone	Orph	Calls	F-ful	F-less	AvgP	Rec
perlbench	1,869	353	34	14,547	6,266	2,083	25	94
bzip2	99	21	4	268	84	58	14	0
gcc	5,562	1,212	281	49,448	21,154	5,190	236	165
mcf	24	11	5	28	12	2	13	0
gobmk	2,679	260	24	10,853	4,753	656	58	6
hmmer	536	74	10	2,222	1,108	232	217	0
sjeng	144	32	4	721	222	139	10	3
libquantum	115	20	4	377	59	126	21	1
h264ref	589	136	43	2,403	1,044	127	863	4
omnetpp	2,833	327	49	12,540	5,567	548	7	1
astar	149	39	15	454	110	49	18	0
xalancbmk	28,601	2,650	90	83,656	16,157	4,087	10	18
Total	43,200	5,135	563	177,517	56,536	13,297		292

Figure 4.11. Constant Propagation (Sec 3.2). **Func**: number of functions. **Clone**: number of clones. **Orph**: number of orphan functions. **Calls**: number of function calls. **F-ful**: number of fruitful calls. **F-less**: number of fruitless calls. **AvgP**: average profit of optimized clone. **Rec**: number of recursive clones.

Figures 4.11–4.13 provide static data to evidence the reach of our optimizations on SPECint. Similarly to what we have done on previous sections, we show their applicability in terms of promising functions. This way, on each table, we show data from a specific clone-based optimization. These data include the number of functions that the benchmark contains (Func), the number of functions that have been cloned (Clone), and the number of functions that became orphans after replacement (Orph). From this data, *we conclude that our optimizations are highly applicable*. In other words, a large proportion of functions are promising (Clone/Func): const-prop = 11.9%, unused-ret-val-elim = 5.8%, pointer disambiguation = 32.9% and fusion = 23.1%. The exception was the elimination of dead stores, that could be applies on only 0.18% of the functions. We already expected the low applicability of the latter optimization, because of the difficulties involved in pinpointing dead memory locations statically.

We also wanted to show the applicability of our optimizations in terms of fruitful call sites, i.e., call sites that can benefit from the optimized clones. So, we show the total number of function calls (Calls) in a given benchmarks, plus the number of fruitful (F-ful) and fruitless sites (F-less). The number of invocations of promising functions is given by F-ful + F-less. The optimization Function Fusion, by definition,

Bench	Func	Clone	Orph	Calls	F-ful	F-less	AvgP	Rec
perlbench	1,869	162	9	14,547	1,984	2,519	176	48
bzip2	99	10	4	268	24	1	3	0
gcc	5,562	272	23	49,448	3,214	6,560	26	45
mcf	24	6	3	28	9	0	0	0
gobmk	2,679	50	4	10,853	1,299	836	10	1
hmmer	536	23	12	2,222	78	51	11	0
sjeng	144	x	x	721	x	x	0	0
libquantum	115	8	3	377	40	22	3	0
h264ref	589	48	19	2,403	243	137	5	1
omnetpp	2,833	118	8	12,540	908	232	2	0
astar	149	7	3	454	8	3	1	0
xalancbmk	28,601	1,817	29	83,656	4,376	2,203	12	13
Total	43,200	2,521	117	177,517	12,183	12,564		108

Figure 4.12. Elimination of unused return values (Sec 3.3).

does not have fruitless sites. The proportion of function calls that are touched by our optimizations (F-ful/Calls) is, again, very high: const-prop = 36.2%, unused-ret-val-elim = 7.4%, pointer disambiguation = 36.7% and fusion = 25.7%. The elimination of dead stores, as an exception, touched just 0.05% of the total number of calls contained in SPECint. In the case of pointer disambiguation, we already expected those results, because, as showed by Mock et al. [2001], the majority of program variables in the SPEC benchmarks point to only a single logical location during execution with the SPEC-provided test inputs.

Figures 4.11–4.13 also provide some interesting numbers from our optimizations. We show the number of cloned functions that are recursive: these functions would pose difficulties to inline expansion, because they cannot be properly inlined. Our clone-based optimizations, otherwise, don't have any difficulties in optimizing this kind of function. In general, a very small proportion of functions fit into this category. For instance, only 3.4% of the clones created by function fusion are recursive. Finally, the tables also provide the average profit that we obtained for each benchmark using the static profiler we have explained in Section 3.7. We calculate the profit of a given clone by the formula $original_cost - cloned_cost$, where $original_cost$ is the static cost from the base function, while $cloned_cost$ is the static cost from the optimized clone. Except for h264ref after pointer disambiguation, the average profits tend to

Bench	Func	Clone	Orph	Calls	F-ful	F-less	AvgP	Rec
perlbench	1,869	292	12	14,547	1,622	0	31	98
bzip2	99	0	0	268	0	0	0	0
gcc	5,562	1,903	43	49,448	20,388	0	17	180
mcf	24	0	0	28	0	0	0	0
gobmk	2,679	24	9	10,853	54	0	6	0
hmmer	536	7	2	2,222	30	0	6	0
sjeng	144	0	0	721	0	0	0	0
libquantum	115	2	1	377	4	0	0	0
h264ref	589	7	7	2,403	28	0	3	0
omnetpp	2,833	214	25	12,540	938	0	5	3
astar	149	5	2	454	14	0	2	0
xalancbmk	28,601	7,550	113	83,656	19,330	0	2	55
Total	43,200	10,004	214	177,517	42,408	0		336

Figure 4.13. Function Fusion (Sec 3.4).

be small, in the order of 50 estimated cycles per function. In the case of `h264ref`, pointer disambiguation has enabled extensive loop unrolling in a video compressing function. Pointer disambiguation is the optimization that yielded the largest static profits. The elimination of unused return values, function fusion and the elimination of dead stores gave us the lowest profits. In the former case, we have observed that many functions simply contain a command `return 0` or `return 1` at its end, which we were eliminating. This return instructions may just indicate if the given function has executed successfully or not. In the case of function fusion, LLVM -O2 has not been able to capitalize much on the extended function bodies that we were creating after function fusion, possible due to the nature of C programs. As for the elimination of dead stores, the removal of store instructions does not seem to generate great opportunities for dead code elimination. Nevertheless, we have observed a few good speedups after reducing the cost of many function by cloning, as we will show in Section 4.3.

Figure 4.16 shows the maximum proportional profits obtained by our optimizations. This number represents the best proportional profit a clone-based optimization could obtain on a given benchmark, and is calculated by the formula $original_cost/cloned_cost$, where $original_cost$ is the static cost from the base function, while $cloned_cost$ is the static cost from the optimized clone. For example, on `perlbench` benchmark, Constant Propagation has selected to clone a function with a

Bench	Func	Clone	Orph	Calls	F-ful	F-less	AvgP	Rec
perlbench	1,869	367	62	14,547	4,031	75	3	2
bzip2	99	19	2	268	34	0	61	0
gcc	5,562	1,769	339	49,448	15,480	241	31	0
mcf	24	7	3	28	7	0	227	0
gobmk	2,679	190	38	10,853	1,337	9	12	0
hmmer	536	160	48	2,222	566	0	1,267	0
sjeng	144	13	3	721	50	1	1	0
libquantum	115	4	1	377	9	0	279	0
h264ref	589	98	44	2,403	764	33	24,057	0
omnetpp	2,833	627	16	12,540	5,211	128	276	0
astar	149	34	13	454	69	1	122	0
xalancbmk	28,601	10,949	632	83,656	30,990	1,005	8	0
Total	43,200	14,237	1,201	177,517	58,548	1,493		2

Figure 4.14. Pointer Disambiguation (Sec 3.5).

Bench	Func	Clone	Orph	Calls	F-ful	F-less	AvgP	Rec
perlbench	1,869	1	0	14,547	2	4	3	1
bzip2	99	0	0	268	0	0	0	0
gcc	5,562	3	1	49,448	6	7	9	2
mcf	24	0	0	28	0	0	0	0
gobmk	2,679	3	0	10,853	3	35	3	0
hmmer	536	4	1	2,222	5	3	3	0
sjeng	144	0	0	721	0	0	0	0
libquantum	115	0	0	377	0	0	0	0
h264ref	589	1	0	2,403	1	32	3	0
omnetpp	2,833	0	0	12,540	0	0	0	0
astar	149	0	0	454	0	0	0	0
xalancbmk	28,601	64	1	83,656	69	44	3	0
Total	43,200	76	3	177,517	86	125		3

Figure 4.15. Elimination of Dead Stores (Sec 3.6).

cost of 312. The optimized clone's cost is 21. This way, the proportional profit of this cloning is $312/21 = 14.85$. We show these numbers as divisions on Figure 4.16, in order to retain the information about static costs.

Bench	Const. Prop.	Elim. Retvals	F. Fusion	Pointer Dis.	Elim. Dead Stores
perlbench	312/21	120/67	4,627/159	78/74	8,780/8,777
bzip2	1,369/774	19/18	–	875/871	–
gcc	88/2	5,295/257	1,563/71	482/179	309/297
mcf	10,169/1,220	122/121	–	210/203	–
gobmk	146/17	39/35	25/20	2,611/2,174	24/21
hmmer	263/17	25/23	144/80	1,041/191	17/17
sjeng	334/127	–	–	44/43	–
libquantum	115/20	13/12	–	1,755/926	–
h264ref	621/15	105/93	611/610	23,174/1,121	–
omnetpp	105/27	65/27	84/42	1,869/259	–
astar	48/34	48/47	–	13,927/1,433	–
xalancbmk	253/15	64/29	81/24	527/73	60/57

Figure 4.16. Maximum proportional profits obtained on SPEC CPU 2006.

We got high proportional profits from both Constant Propagation and Pointer Disambiguation. Function Fusion also got nice profits on `perlbench` and `gcc`. Maximum profits for Elimination of Dead Stores and Elimination of Unused Retvals tend to be small, with an exception being the latter optimization on `gcc` benchmark. We will study some of the functions with high proportional profits on Chapter 5.

The main conclusion that we draw from these tables is that our clone-based optimizations are widely applicable and are able to reduce the costs of functions, even in mature programs, such as those found in SPEC CPU 2006.

4.3 Effectiveness: Time and Space

In this section, we analyze how our proposed clone-based optimizations affect compilation time, runtime and code size. We have made our tests on the same benchmarks used in Section 4.2: SPEC CPU 2006 and LLVM Test-Suite. We will analyze each of these collections individually.

When analyzing runtime, we have run each test 15 times. Our runtimes are compared against the optimization level LLVM -O2.

4.3.1 Test-Suite SingleSource

Compilation time. LLVM provides an easy way to see how much time an optimization step took during the compilation². Figure 4.17 shows the average percentage of time each of our optimizations took to run on Test-Suite SingleSource benchmarks.

Optimization	Average % of Compilation Time
Constant Propagation	1.15%
Elimination of Unused Return Values	0.54%
Function Fusion	0.49%
Pointer Disambiguation	0.70%

Figure 4.17. Compilation time of our optimizations when applied to Test-Suite SingleSource.

From Figure 4.17, we can see that our optimizations do not take a large share of the compilation time. Actually, our slowest optimization, Constant Propagation, accounted for an average of 1.15% of total compilation time. The time required to run the other optimizations were less than 1% of total time. Thus, we can say that the proposed clone-based optimizations are safe to be used on programs that require short compilation times.

Runtime. In order to analyze the runtime improvements provided by our optimizations, we have chosen to calculate the speedups obtained on the biggest benchmarks contained in Test-Suite SingleSource. We got the 20 largest benchmarks by choosing the ones that take more time to run. Figure 4.18 shows the runtime variations for these benchmarks when applying our optimizations. We are comparing our runtimes against the ones obtained with the LLVM `-O2` optimization level. Elimination of dead stores is not shown in this figure, because it found no optimization opportunities on these benchmarks.

Constant propagation got good results on *lists*, with an improvement of 4.37%. Elimination of unused retvals yielded an improvement of 2.02% on *ReedSolomon*. Function fusion got nice numbers on *almabench* (4.07%) and *spirit* (3.97%). Pointer disambiguation was our best optimization, yielding 24.5% of improvement on top of *loop_unroll*, and good results on *smallpt* (13.38%) and *spirit* (10.48%).

Space. In order to analyze how our optimizations affect the size of programs, we have measured the size of the binaries generated at the end of the pipeline shown in Figure 4.1. We used the UNIX command `wc -c` to get the size of the binaries.

²Documented on-line at <http://llvm.org/docs/CommandGuide/llc.html#cmdoption-time-passes>.

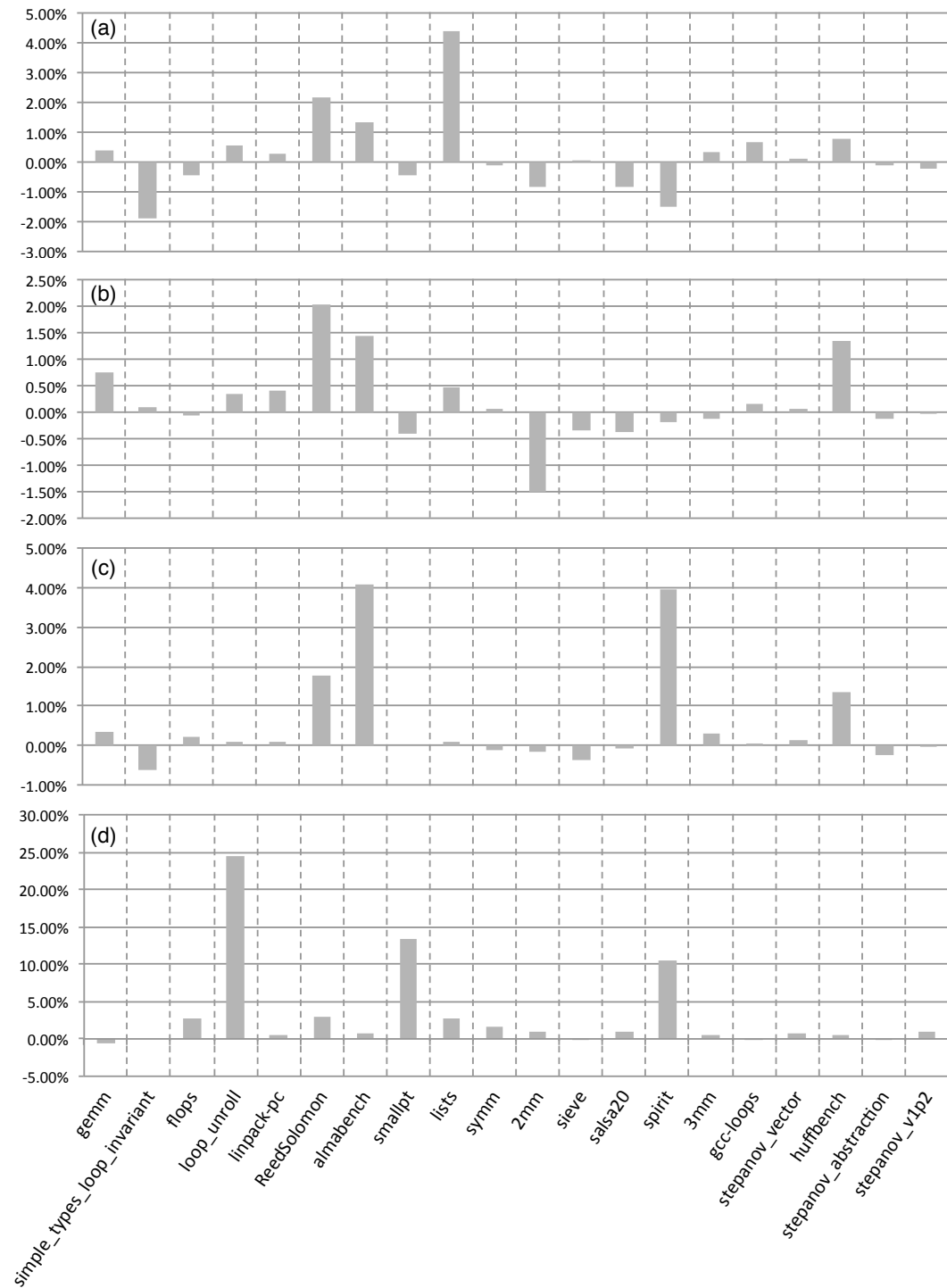


Figure 4.18. Runtime variations on Test-Suite SingleSource. (a) Constant propagation. (b) Elimination of unused return values. (c) Function fusion. (d) Pointer disambiguation.

We got the sizes of the programs without inline expansion on the polishing step, like we explained on Section 4.1. We did this because inlining could fool our results, as it would expand calls for many of our cloned functions. Figure 4.19 shows how our optimizations affects the size of the entire Test-Suite SingleSource collection. We also show how inline expansion contributes to code increase.

Optimization	% of increase in size
Constant Propagation	1.54%
Elimination of Unused Return Values	0.23%
Function Fusion	2.77%
Pointer Disambiguation	1.18%
Inline Expansion	6.47%

Figure 4.19. Size increase on binaries in Test-Suite SingleSource.

Inline expansion increased the code size of Test-Suite SingleSource benchmarks in 6.47%. Our optimizations were less expensive. The clone-based optimization that yielded the largest code size increase was function fusion, with 2.77% of code growth. The use of all other optimizations resulted in less than 2% of code size growth.

4.3.2 Test-Suite MultiSource

Compilation time. Figure 4.20 shows the average percentage of time each of our optimizations took to run on the benchmarks of Test-Suite MultiSource.

From Figure 4.20, we see that, even on large programs, like the ones contained on Test-Suite MultiSource, our optimizations don't take much time to run. Our slowest optimization on this collection of benchmarks was Constant Propagation, accounting

Optimization	Average % of Compilation Time
Constant Propagation	2.36%
Elimination of Unused Return Values	0.82%
Function Fusion	0.28%
Pointer Disambiguation	0.80%
Elimination of Dead Stores	0.15%

Figure 4.20. Compilation time of our optimizations when applied to Test-Suite MultiSource.

for an average of 2.36% of total compilation time. The other optimizations took less than 1% of total compilation time to run.

Runtime. Figure 4.21 shows the runtime variations for each of our optimizations on the biggest benchmarks of Test-Suite MultiSource. We are comparing our optimized programs with the ones obtained after the application of the LLVM -O2 optimization level.

Most runtime improvements stayed between -1% and 1%. We consider this variations as inconclusive results. Our best result was obtained when applying pointer disambiguation optimization on top of *enc-pc1*, that yielded 10.39% of improvement. Constant propagation got an improvement of 3.02% on *pairlocalalign*. Function fusion could speed up *minisat* in 1.8%.

Space. Figure 4.22 shows the code size growth caused by the application of our optimizations. For comparison purposes, we also show how inline expansion affects code size.

On Test-Suite MultiSource, the application of constant propagation and function fusion resulted in large code growths, 15.08% and 10.26%, respectively. These results were really close to inline expansion, that increased the code size in 14.22%. Elimination of unused retvals and elimination of dead stores didn't increase the code by much, as they were less applicable on the benchmarks. Pointer disambiguation yielded a slightly code growth, 6.09%.

The applicability charts on Section 4.2.2 can help us understand why the application of constant propagation resulted in a significant code growth. This optimization is highly applicable on the benchmarks of MultiSource Test-Suite. Besides that, most clones are not able to replace all the call sites from their base functions, i.e., it is uncommon that all the call sites of a given function are fruitful. This way, the final code often contains two versions of each promising function: the base and the clone. Had the clone the ability to replace all the call sites, the base function would become orphan and would be eliminated on our pipeline, just like we explained on Section 4.1.

As for function fusion, the definition of the optimization makes its application more likely to result in a big code growth: we are combining the bodies of two functions under a new function, i.e., the clone. It is really rare a situation where the clone can replace all the call sites from both functions used to created it. This will only happen if these two functions are always used together, in a way that all their call sites can be combined. As we can see from the applicability charts on Section 4.2.2, this situation is very unlikely to happen.

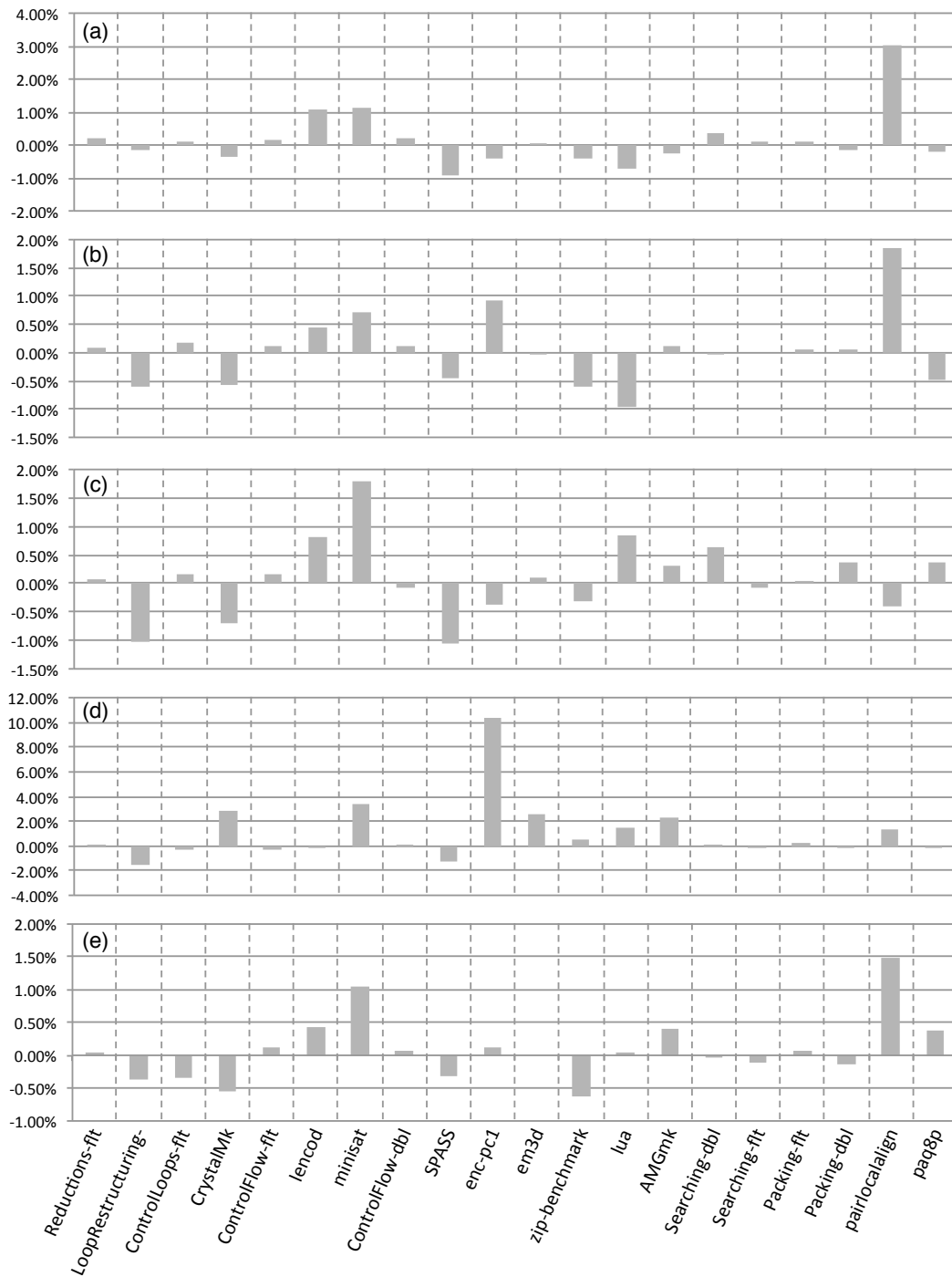


Figure 4.21. Runtime variations on Test-Suite MultiSource. (a) Constant propagation. (b) Elimination of unused return values. (c) Function fusion. (d) Pointer disambiguation. (e) Elimination of dead stores.

Optimization	% of increase in size
Constant Propagation	15.08%
Elimination of Unused Return Values	2.58%
Function Fusion	10.26%
Pointer Disambiguation	6.09%
Elimination of Dead Stores	0.18%
Inline Expansion	14.22%

Figure 4.22. Size increase on binaries in Test-Suite MultiSource.

4.3.3 SPEC CPU 2006

Compilation time. Figure 4.23 shows the average percentage of time our optimizations took to run on the benchmarks of SPEC. Even on large benchmarks, our optimizations did not present any harm for compilation time. Constant propagation was our slowest optimization, accounting for an average of 1.39% of compilation time. The other optimizations took less than 1% of compilation time to be applied.

Optimization	Average % of Compilation Time
Constant Propagation	1.39%
Elimination of Unused Return Values	0.29%
Function Fusion	0.52%
Pointer Disambiguation	0.91%
Elimination of Dead Stores	0.17%

Figure 4.23. Compilation time of our optimizations when applied to SPEC CPU 2006.

Runtime. The charts in Figure 4.24 show the runtime variation that our optimizations produce for all the SPEC CPU 2006 programs. This time we show results for the integer and floating point benchmarks. We regard runtime variations of less than 1% as inconclusive. Our largest improvement was 5.9% due to pointer disambiguation on `483.xalancbmk`. Function fusion gave us good results in `403.gcc` (3.3%). We did not get expressive results due to the elimination of unused return values nor elimination of dead stores.

Both inlining and clone-based optimizations perform context-aware optimizations. Thus, we also compared the effects in runtime that the application of all our clone-based optimizations together could achieve against inline expansion. In order to

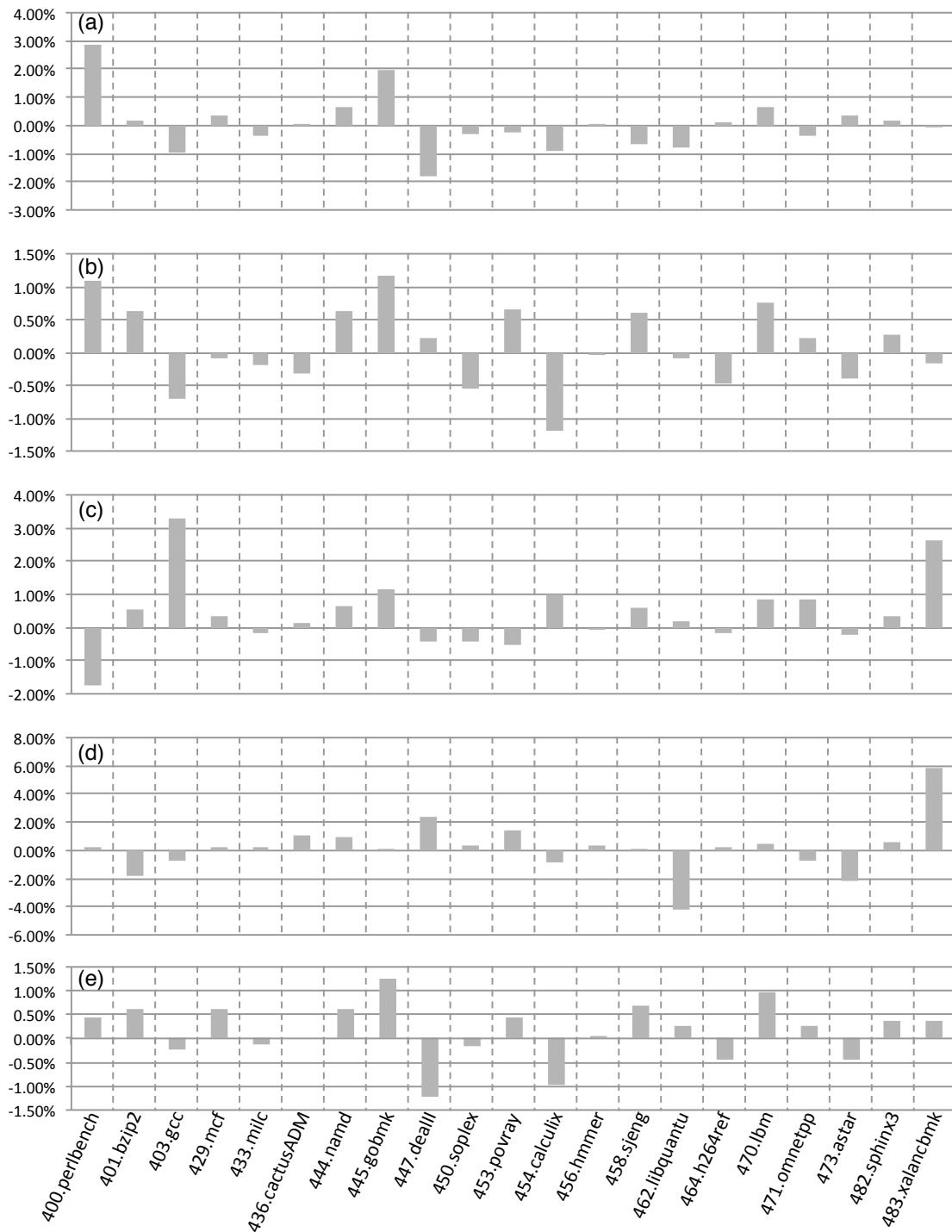


Figure 4.24. Runtime variations on SPEC CPU 2006 (a) Constant propagation. (b) Elimination of unused return values. (c) Function fusion. (d) Pointer disambiguation. (e) Elimination of dead stores.

do that, we compiled the benchmarks on SPEC with three different sets of optimizations described as follows:

- (i) *Baseline* (-O2 without inlining);
- (ii) *Inlining* (full -O2);
- (iii) *Cloning* (all clone-based optimizations together, excluding inline expansion from the polishing step).

We then calculated the runtime improvements obtained with (ii) and (iii) when compared to the runtimes of (i). Figure 4.25 shows the improvements obtained on each benchmark of SPEC when using the set of optimizations (ii) and (iii)

Benchmark	<i>Inlining</i> runtime improvement	<i>Cloning</i> runtime improvement
400.perlbench	1.20%	1.55%
401.bzip2	0.94%	0.14%
403.gcc	6.27%	1.51%
429.mcf	3.90%	0.95%
433.milc	0.80%	0.96%
436.cactusADM	0.25%	0.54%
444.namd	4.69%	0.01%
445.gobmk	0.68%	0.50%
447.dealII	413.83%	14.37%
450.soplex	43.11%	-0.51%
453.povray	60.97%	-0.08%
454.calculix	0.06%	-0.76%
456.hmmer	0.15%	-0.17%
458.sjeng	3.64%	-0.73%
462.libquantum	4.29%	2.97%
464.h264ref	0.94%	0.82%
470.lbm	-0.32%	-0.22%
471.omnetpp	65.81%	2.86%
473.astar	21.86%	-2.01%
482.sphinx3	1.52%	-0.43%
483.xalancbmk	307.24%	7.26%

Figure 4.25. Runtime improvements of all our clone-based optimizations versus LLVM -O2

The *Cloning* set of optimizations got some good results when applied on top of -O2 without inlining. It was able to speed up `447.dealII` in 14.37% and `483.xalancbmk` in 7.26%. However, this set of optimizations could not beat *Inlining* in runtime improvement on most SPEC CPU 2006 benchmarks. This happens because inline expansion is able to apply context-aware optimizations extensively, while our clone-based approach implements just predefined optimizations.

Space. One of the advantages cloning provides on top of inline expansion is that it generates reusable code. Thus, a clone is an optimized function that can be invoked at different call sites. Inline expansion, on the other hand, requires every call

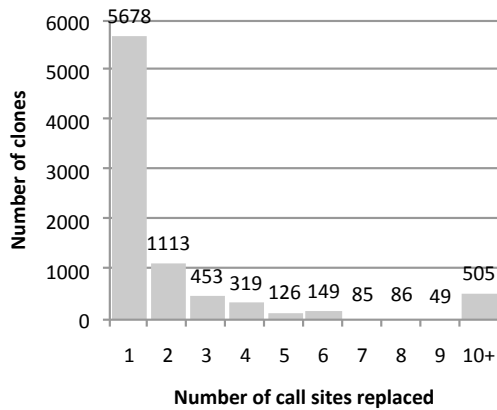


Figure 4.26. Constprop: Fruitful sites

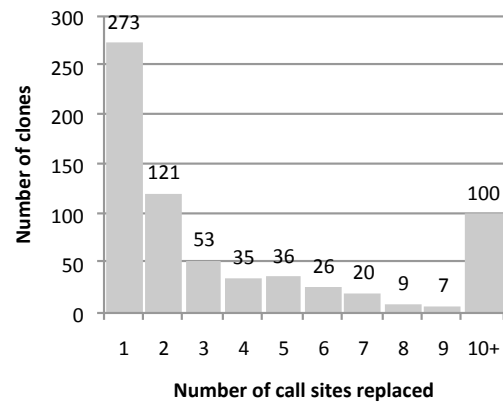


Figure 4.27. Elim. of Unused Retvals: Fruitful sites

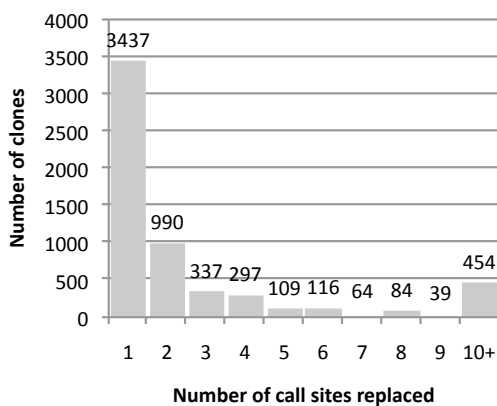


Figure 4.28. Function Fusion: Fruitful sites

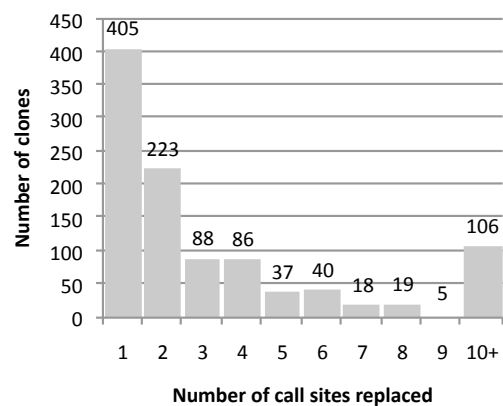


Figure 4.29. Pointer Dis.: Fruitful sites

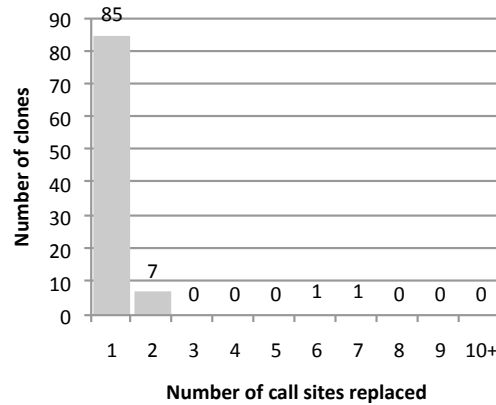


Figure 4.30. Elimination of Dead Stores: Number of fruitful sites

site to be replaced with the called function’s body, producing a large code expansion. We have analyzed the clones generated by the application of each of our clone-based optimization on SPEC. We have counted the number of fruitful call sites each clone was able to touch. These data are shown on Figures 4.26–4.30 as histograms. We got these numbers after applying our optimizations without inline expansion on the polishing step.

We can see that a large number of clones are invoked at different call sites. In fact, 60.56% of the clones generated by Pointer Disambiguation are called in more than one site. The other optimizations generated proportions of clones called more than once of 59.85% for Elimination of Unused Return Values, 42.1% for Function Fusion, 33.69% for Constant Propagation and 9.57% for Elimination of Dead Stores.

Table 4.31 shows some interesting numbers about our optimizations. It shows the maximum number of times a cloned function is called, i.e., the maximum number of fruitful call sites a single cloning procedure could optimize. The *gcc* benchmark generated good opportunities for our optimizations. In fact, a single clone created by Constant Propagation was able to optimize 1,268 call sites in *gcc*. Function Fusion also created a clone able to optimize 1,873 sites. Cloning with Pointer Disambiguation yielded an optimized function that could replace 405 fruitful sites on the same benchmark.

Figure 4.32 shows the code growth generated by our clone-based optimizations on SPEC CPU 2006. We have calculated the percentage of size growth in all the compiled binaries. In order to do that, we have compared the sizes after the optimization pipeline against the sizes of the binaries compiled with the set of optimizations `-O2` without inlining. We also show the code growth caused by the application of inline expansion.

From 4.32, we conclude that our clone-based approach produces less code ex-

Benchmark	Constprop	Retvals elim.	Fusion	Pointer dis.	Elim. dead stores
perlbench	161	76	67	308	2
bzip2	5	4	0	1	0
gcc	1286	669	1873	405	2
mcf	2	1	0	1	0
milc	8	22	0	46	0
cactusADM	17	727	9	16	1
namd	3	0	3	21	0
gobmk	215	43	2	19	1
dealII	203	115	334	86	0
soplex	37	2	18	5	1
povray	274	166	52	27	2
calculix	54	80	9	114	7
hmmer	38	36	5	18	2
sjeng	15	0	0	6	0
libquantum	4	3	1	3	0
h264ref	13	24	5	139	1
lbm	1	0	0	1	0
omnetpp	399	113	18	36	0
astar	8	1	2	7	0
sphinx3	20	2	1	7	2
xalancbmk	85	74	45	206	6

Figure 4.31. Maximum number of replaced call sites

pansion than inlining. Our optimization that yielded the largest code size growth was Function Fusion, with 24.85% of increase. Inline expansion, on the other hand, increased code size in 57.45%.

The tests shown above were made using LLVM’s implementation of inline expansion. This implementation tends to inline just small functions, and procedures that are called only once. Our clone-based approach, on the other hand, is applied extensively. To provide a fairer comparison, we have made some tests using the following methodology: for each non-recursive promising function, we replicate its body at each fruitful site, and record the number of new instructions created. The sum of new instructions created for each function is the size of the program, after inlining, which we denote by I . We let C be the size of the program after cloning, i.e., the number

Optimization	% of increase in size
Constant Propagation	17.53%
Elimination of Unused Return Values	1.80%
Function Fusion	24.85%
Pointer Disambiguation	4.00%
Elimination of Dead Stores	0.17%
Inline Expansion	57.45%

Figure 4.32. Size increase on binaries in SPEC CPU 2006.

of new instructions created by cloning. Figure 4.33 reports the ratio C/I . Sizes are as following: Constant propagation: $I = 16,714,965$, $C = 922,043$. Elimination of unused returns: $I = 3,306,904$, $C = 201,359$. Pointer disambiguation: $I = 1,441,569$, $C = 603,573$. Function fusion: $I = 36,362,025$, $C = 1,856,044$. Elimination of dead stores: $I = 46,569$, $C = 11,097$. Thus, we conclude that inlining produces one order of magnitude more instructions than our clone-based optimizations.

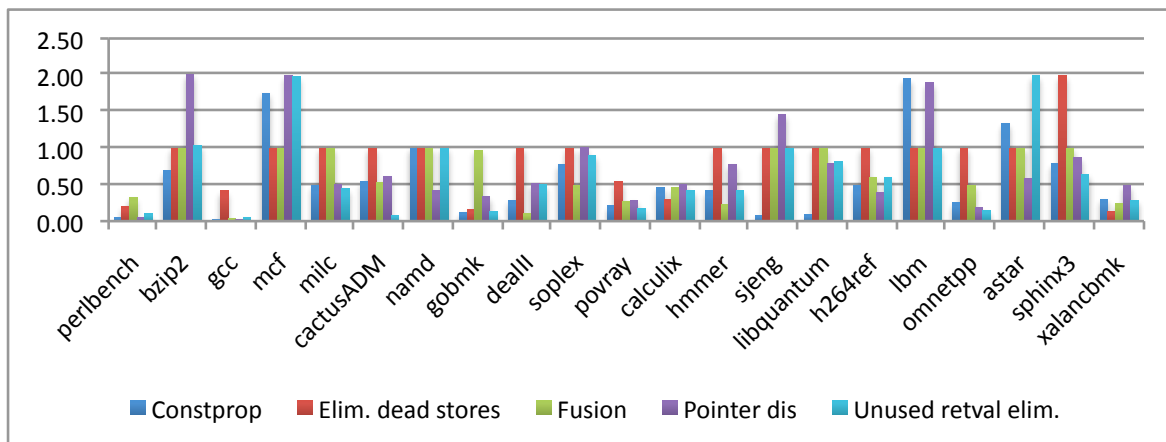


Figure 4.33. Size reduction due to our optimizations.

4.4 Discussion

In this chapter, we have provided data to show the applicability and effectiveness of our clone-based approach. The application of the proposed optimizations has led to minimal increases in compilation times. Thus, their use is appropriate even in large programs. Considering the runtime impact of our optimizations, we have been able

to speedup a large amount of benchmarks, ranging from small programs to large industrial applications. Among the novel optimizations described in this work, Pointer Disambiguation has led to our best results. In particular, it was able to improve the `loop_unroll` benchmark by 24.5%, reducing its runtime in almost one quarter. In general, we have not been able to observe large speedups in the SPEC CPU 2006 programs. This happens because these benchmarks come from very mature applications.

When comparing inline expansion with cloning without inlining on the *polishing* step, the former got the best runtime results. Inline expansion is a strong enabler of context-aware optimizations, as it does not rely on predefined optimizations like cloning does. Cloning, on the other hand, has the advantages of leading to less code expansion and generating reusable functions.

Chapter 5

Case Studies

In this chapter, we dive into the Software Engineering side of our clone-based approach. We analyze some functions found on known benchmarks and applications in the light of the benefits provided by our optimizations. We try to understand how and how much the proposed optimizations can improve the compiled code, using the static cost methodology presented in Section 3.7.

In order to study how our clone-based optimizations can enhance the compiled code, we got the functions that yielded the best profits on each benchmark. We then analyzed these functions, studying its original source code and intermediate code, in order to find out why the application of a given optimization generated such a great profit. We present, in the following sections, the functions we found to be of great interesting.

5.1 The Fft Function

Our experiments indicate that Pointer Disambiguation is the most effective optimization in our suite of techniques. It was able to generate the best speedups when applied on a variety of benchmarks. As we explained on Section 3.5, Pointer Disambiguation tells the compiler that a pointer, passed as parameter to a given function, has no aliases on the function's context. This way, the compiler is able to perform more aggressive optimizations on the code. In this section, we will show the benefits of applying Pointer Disambiguation to a function named `Fft`, shown in Figure 5.1.

The `Fft` function is present on one of the Stanford Benchmarks¹. Among its parameters, this function receives three arrays of the same struct, namely `z`, `w` and `e`.

¹ <https://llvm.org/viewvc/llvm-project/test-suite/trunk/SingleSource/Benchmarks/Stanford/>

```

1  struct complex {
2      float rp, ip;
3  };
4  void Fft (int n, struct complex z[], struct complex w[],
5  struct complex e[], float sgrinv) {
6      int i, j, k, l, m, index;
7      m = n / 2;
8      l = 1;
9      do {
10         k = 0;
11         j = 1;
12         i = 1;
13         do {
14             do {
15                 w[i + k].rp = z[i].rp + z[m + i].rp;
16                 w[i + k].ip = z[i].ip + z[m + i].ip;
17                 w[i + j].rp = e[k + 1].rp * (z[i].rp
18                     - z[i + m].rp) - e[k + 1].ip * (z[i].ip - z[i + m].ip);
19                 w[i + j].ip = e[k + 1].rp * (z[i].ip
20                     - z[i + m].ip) + e[k + 1].ip * (z[i].rp - z[i + m].rp);
21                 i = i + 1;
22             }
23             while (i <= j);
24             k = j;
25             j = k + 1;
26         }
27         while (j <= m);
28         index = 1;
29         do {
30             z[index] = w[index];
31             index = index + 1;
32         }
33         while (index <= n);
34         l = l + 1;
35     }
36     while (l <= m);
37     for (i = 1; i <= n; i++) {
38         z[i].rp = sgrinv * z[i].rp;
39         z[i].ip = -sgrinv * z[i].ip;
40     }
41 }

```

Figure 5.1. Fft function present on Stanford Benchmarks.

During its main while loop, the function changes the values of the elements of w based on the elements of z and e , between lines 14 and 23.

The compiler must be conservative and assume, by default, that any of z , w and e can alias each other, i.e., any of these pointers can point to the same memory location. This way, the compiler has to load the values of $z[i]$ and $e[i]$ from memory whenever they are used on lines 14..23, because the previous assignment to $w[i]$ may have changed their values. If the vectors cannot alias, however, the compiler can cache the values of $z[i]$ and $e[i]$, loading them only once during an iteration. This is possible because neither $z[i]$ nor $e[i]$ appears on the left-hand side of any instruction during an iteration. This way, the application of Pointer Disambiguation generates an optimized code like the one shown in Figure 5.2.

We show the optimized source code in C for an easier understanding. Our opti-

```

1 void Fft.noalias (int n, struct complex restrict z[],
2 struct complex restrict w[], struct complex restrict e[], float sqrinv) {
3 (...)
4 do {
5     zirp = z[i].rp;
6     ziip = z[i].ip;
7     zmirp = z[m + i].rp;
8     zmiip = z[m + i].ir;
9     zimrp = z[i + m].rp;
10    zimip = z[i + m].ir;
11    ekrp = e[k + 1].rp;
12    ekip = e[k + 1].ip;
13
14    w[i + k].rp = zirp + zmirp;
15    w[i + k].ip = ziip + zmiip;
16    w[i + j].rp = ekrp * (zirp - zimrp) - ekip * (ziip - zimip);
17    w[i + j].ip = ekrp * (ziip - zimip) + ekip * (zirp - zimrp);
18    i = i + 1;
19 }
20 while (i <= j);
21 (...)

```

Figure 5.2. Function `Fft` optimized with Pointer Disambiguation.

mization is, however, applied on the intermediate code representation. When analyzing the intermediate code of the base and cloned versions, we see that Pointer Disambiguation is indeed able to remove memory loads. The base version of `Fft`, optimized with the `-O2` optimization level, contains 19 load instructions. The cloned version, in contrast, has had half of these instructions eliminated, presenting just 8 load instructions. When analyzing both the base and cloned versions in terms of static cost, we obtained costs of 30500 and 21800, respectively. So, we see that Pointer Disambiguation could reduce the static function cost in almost 30%.

5.2 The `dmxpy` function

`Dmxpy` is another function we will use to show the benefits of Pointer Disambiguation. This function, shown in Figure 5.3, is present on the Linpack Benchmark ², known for testing performance of floating point operations.

The application of Pointer Disambiguation on function `dmxpy` makes the compiler able to hoist many load instructions out of `for` loop bodies. The `dmxpy` has two vector, `x` and `y` among its parameters. The nested `for` loops between lines 7 and 26 are responsible for changing the values of `y` based on values of `x` and others. More specifically, each new value assigned to an element of `y` depends on the elements of `x` from `x[j-15]` to `x[j]`. This way, considering `x` and `y` can alias each other, the compiler has to load these 16 elements every time the assignment of line 9 is made. This happens because the present assignment can change the value of any element of `x`. Thus, the execution

²<http://www.top500.org/project/linpack/>

```

1 void dmxpy (int n1, double y[], int n2, int ldm, double x[], double m[]) {
2     int j, i, jmin;
3
4     (...)
5
6     jmin = (n2%16)+16;
7     for (j = jmin-1; j < n2; j = j + 16) {
8         for (i = 0; i < n1; i++)
9             y[i] = ((((((((((((((y[i])
10                + x[j-15]*m[ldm*(j-15)+i])
11                + x[j-14]*m[ldm*(j-14)+i])
12                + x[j-13]*m[ldm*(j-13)+i])
13                + x[j-12]*m[ldm*(j-12)+i])
14                + x[j-11]*m[ldm*(j-11)+i])
15                + x[j-10]*m[ldm*(j-10)+i])
16                + x[j-9]*m[ldm*(j-9)+i])
17                + x[j-8]*m[ldm*(j-8)+i])
18                + x[j-7]*m[ldm*(j-7)+i])
19                + x[j-6]*m[ldm*(j-6)+i])
20                + x[j-5]*m[ldm*(j-5)+i])
21                + x[j-4]*m[ldm*(j-4)+i])
22                + x[j-3]*m[ldm*(j-3)+i])
23                + x[j-2]*m[ldm*(j-2)+i])
24                + x[j-1]*m[ldm*(j-1)+i])
25                + x[j] *m[ldm*j+i]);
26     }
27     return ;
28 }

```

Figure 5.3. dmxpy function present on Linpack Benchmark.

has to emit 16 load instructions before computing and assigning the new value to an element of y .

If vectors x and y do not alias each other, the loads of elements of x become invariants in the inner loop. They can, therefore, be hoisted from the inner to the outer loop body. This way, the number of load instructions executed reduce in $n1$ times. Figure 5.4 show how the optimized code would look if it was written in C.

When analyzing the static costs of the base and cloned versions of `dmxpy`, we see that Pointer Disambiguation was able to reduce the cost from 6050 to 5310, an improvement of 12%.

5.3 The readULONG Function

One of the many benchmarks contained in SPEC CPU 2006 is povray. This benchmark has a small function responsible for reading a given file in chunks of 4 bytes. These bytes are then combined in a unique byte using shift operations. The result of these operations is returned to the caller. The source code of this function, named `readULONG` is shown on Figure 5.5.

When reading a given file, however, the bytes corresponding to its header may not be interesting to the programmer, as they only contain meta-data about the file.

```

1 void dmxpy.noalias (int n1, double restrict y[],
2 int n2, int ldm, double restrict x[], double m[]) {
3     int j, i, jmin;
4
5     (...)
6
7     jmin = (n2%16)+16;
8     for (j = jmin-1; j < n2; j = j + 16) {
9         xj = x[j];
10        xj1 = x[j- 1];
11        xj2 = x[j- 2];
12        xj3 = x[j- 3];
13        xj4 = x[j- 4];
14        xj5 = x[j- 5];
15        xj6 = x[j- 6];
16        xj7 = x[j- 7];
17        xj8 = x[j- 8];
18        xj9 = x[j- 9];
19        xj10 = x[j-10];
20        xj11 = x[j-11];
21        xj12 = x[j-12];
22        xj13 = x[j-13];
23        xj14 = x[j-14];
24        xj15 = x[j-15];
25        for (i = 0; i < n1; i++)
26            y[i] = ((((((((((((((y[i]
27                + xj15*m[ldm*(j-15)+i])
28                + xj14*m[ldm*(j-14)+i])
29                + xj13*m[ldm*(j-13)+i])
30                + xj12*m[ldm*(j-12)+i])
31                + xj11*m[ldm*(j-11)+i])
32                + xj10*m[ldm*(j-10)+i])
33                + xj9 *m[ldm*(j- 9)+i])
34                + xj8 *m[ldm*(j- 8)+i])
35                + xj7 *m[ldm*(j- 7)+i])
36                + xj6 *m[ldm*(j- 6)+i])
37                + xj5 *m[ldm*(j- 5)+i])
38                + xj4 *m[ldm*(j- 4)+i])
39                + xj3 *m[ldm*(j- 3)+i])
40                + xj2 *m[ldm*(j- 2)+i])
41                + xj1 *m[ldm*(j- 1)+i])
42                + xj *m[ldm*j+i]);
43    }
44    return;
45 }

```

Figure 5.4. dmxpy function optimized with Pointer Disambiguation.

This happens on the povray benchmark. In this case, the source code makes a call to the `readULONG` function, discarding its return value, i.e, the caller does not use a variable to store the value returned by the callee. In this situation, the Elimination of Unused Return Values can be used to improve the code generated by the compiler. The application of this clone-based optimization would yield the code shown in Figure 5.6 if it was applied on C code.

We can see that the optimized version of `readULONG` doesn't execute any shift operations with the bytes read on lines 5–8, as the result of these operations are not interesting to the caller function. When analyzing the base and cloned versions of `readULONG` using the static profiler, we see that our proposed optimization could reduce

```

1  static ULONG readULONG(IStream *infile, int line,
2  const char *file) {
3      int i0, i1 = 0, i2 = 0, i3 = 0;
4
5      if ((i0 = infile->Read_Byte ()) == EOF ||
6          (i1 = infile->Read_Byte ()) == EOF ||
7          (i2 = infile->Read_Byte ()) == EOF ||
8          (i3 = infile->Read_Byte ()) == EOF) {
9          Error("Error reading TrueType font file at line %d, %s",
10             line, file);
11     }
12
13     return (ULONG) (((ULONG) i0) << 24) |
14                (((ULONG) i1) << 16) |
15                (((ULONG) i2) << 8) |
16                ((ULONG) i3));
17 }

```

Figure 5.5. readULONG function present on povray benchmark.

```

1  static void readULONG.noret(IStream *infile, int line,
2  const char *file) {
3      int i0, i1 = 0, i2 = 0, i3 = 0;
4
5      if ((i0 = infile->Read_Byte ()) == EOF ||
6          (i1 = infile->Read_Byte ()) == EOF ||
7          (i2 = infile->Read_Byte ()) == EOF ||
8          (i3 = infile->Read_Byte ()) == EOF) {
9          Error("Error reading TrueType font file at line %d, %s",
10             line, file);
11     }
12
13     return;
14 }

```

Figure 5.6. readULONG function optimized with Elimination of Unused Retvals.

the cost of this function from 23 to 9, a reduction of 60%.

Chapter 6

Final Remarks

6.1 Conclusion

The goal of this dissertation was to show that code specialization using cloning can be effective in implementing context-aware optimizations. We have used the definition of context-aware optimizations to find performance bugs that could be removed with cloning. Then, we have designed optimizations that were able to remove those bugs. We have also discussed a way to estimate the profitability of a clone, based on Wu and Larus's static profiler. Our optimizations, although simple, have wide applicability. We have been able to speedup some known benchmarks up to 25%. We believe that function cloning is a useful and effective tool to fight performance bugs. In this sense, cloning serves programmers better than inlining, because it gives them units of reusable code - the cloned functions.

Software: In page <https://github.com/matheusvilela/clone-based-opts> you may find the code for all our clone-based optimizations, plus our static profiler.

6.2 Future Work

As future work, we want to study better the relationship between cloning and inlining. For instance, we want to be able to decide if a given function should be cloned or inlined during the optimization step. In order to decide which optimization gives us the greatest benefit, we aim to use the same static profiler presented in this work.

We also want to improve our framework in order to be able to warn about a possible performance bug to the developer. We want to do this using the static profiler results. If the cost of a cloned function is much lower than the original's cost, the implementation of this function may have a performance bug.

Bibliography

- Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen.
- Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.
- Bernstein, D., Cohen, D., and Maydan, D. E. (1994). Dynamic memory disambiguation for array references. In *Proceedings of the 27th Annual International Symposium on Microarchitecture, MICRO 27*, pages 105–111, New York, NY, USA. ACM.
- Blanchet, B. (1998). Escape analysis: Correctness proof, implementation and experimental results. In *POPL*, pages 25–37. ACM.
- Bolat, M. and Li, X. (2009). Context-aware code optimization. *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International Computers, IEEE Transactions*, pages 256–263.
- Chabbi, M. and Mellor-Crummey, J. (2012). Deadspy: a tool to pinpoint program inefficiencies. In *CGO*, pages 124–134. ACM.
- Chang, P. P. and Hwu, W. W. (1988). Trace selection for compiling large c application programs to microcode. *MICRO 21: Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, pages 21–29.
- Chin, W.-N. (1992). Safe fusion of functional expressions. In *LFP*, pages 11–20. ACM.
- Consel, C. and Noël, F. (1996). A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96*, pages 145–156, New York, NY, USA. ACM.
- Cooper, K. D., Hall, M. W., and Kennedy, K. (1993). A methodology for procedure cloning. *Comput. Lang.*, 19(2):105–117. ISSN 0096-0551.

- Cooper, K. D., Hall, M. W., and Torczon, L. (1991). An experiment with inline substitution. *Softw. Pract. Exper.*, 21(6):581--601. ISSN 0038-0644.
- de Assis Costa, I. R., Alves, P. R. O., Santos, H. N., and Pereira, F. M. Q. (2013). Just-in-time runtime specialization. In *CGO*, pages 1--11. ACM.
- Fisher, J. (1981). Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions*, C-30(7):478--490.
- Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. (2000). Dyc: An expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci.*, 248(1-2):147--199. ISSN 0304-3975.
- Grune, D., van Reeuwijk, K., Jacobs, H. E. B. C. J. H., and Langendoen, K. (2012). *Modern Compiler Design*. Springer, 2nd edition.
- Hall, M. W. (1991). *Managing interprocedural optimization*. PhD thesis, Rice University, Houston, TX, USA. UMI Order No. GAX91-36029.
- Hardekopf, B. and Lin, C. (2007). The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290--299. ACM.
- Hennessy, J. L. and Patterson, D. A. (2003). *Computer Architecture: A Quantitative Approach*. Elsevier, 3rd edition.
- Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1--17.
- Holmes, R. and Walker, R. J. (2013). Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4):20:1--20:44.
- Huang, D. (2011). Programmer-assisted automatic parallelization. Master's thesis, University of Toronto.
- ISO (1999). ISO C standard 1999. Technical report, American National Standards Institute. ISO/IEC 9899:1999 draft.
- Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. (2012). Understanding and detecting real-world performance bugs. In *PLDI*, pages 77--88. ACM.
- Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-020249-5.

- Jovic, M., Adamoli, A., and Hauswirth, M. (2011). Catch me if you can: performance bug detection in the wild. In *OOPSLA*, pages 155–170. ACM.
- Keith D. Cooper, L. T. (2012). *Engineering a Compiler*. Morgan Kaufmann, 2nd edition.
- Kennedy, K. and Allen, J. R. (2002). *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-286-0.
- Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2):131--183.
- Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.
- Lau, J., Arnold, M., Hind, M., and Calder, B. (2006). Online performance auditing: using hot optimizations without getting burned. *SIGPLAN*, 41(6):239–251.
- Lhoták, O. and Hendren, L. (2006). Context-sensitive points-to analysis: is it worth it? In *CC*, pages 47--64. Springer-Verlag.
- Metzger, R. and Stroud, S. (1993). Interprocedural constant propagation: an empirical study. *ACM Lett. Program. Lang. Syst.*, 2(1-4):213--232.
- Mock, M. (2004). Why programmer-specified aliasing is a bad idea. In *CLEI*, pages On-line. SCEAS.
- Mock, M., Das, M., Chambers, C., and Eggers, S. J. (2001). Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pages 66--72, New York, NY, USA. ACM.
- Nistor, A., Song, L., Marinov, D., and Lu, S. (2013). Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 1–10. IEEE.
- Oracle (2013). *C99 Keyword (Sun Studio 12 Update 1: C User's Guide)*.
- Samadi, M., Hormati, A., Mehrara, M., Lee, J., and Mahlke, S. (2012). Adaptive input-aware compilation for graphics engines. In *PLDI*, pages 13--22. ACM.
- Sedgewick, R. (1984). Algorithms. In *Algorithms*, page 84.

- Shafer, G. (1976). *A Mathematical Theory of Evidence*. Princeton University Press, 1st edition.
- Shankar, A., Sastry, S. S., Bodík, R., and Smith, J. E. (2005). Runtime specialization with optimistic heap analysis. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 327--343, New York, NY, USA. ACM.
- Shivers, O. (1988). Control-flow analysis in scheme. In *PLDI*, pages 164--174. ACM.
- St-Amour, V., Tobin-Hochstadt, S., and Felleisen, M. (2012). Optimization coaching: optimizers learn to communicate with programmers. In *OOPSLA*, pages 163--178. ACM.
- Steensgaard, B. (1996). Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32--41, New York, NY, USA. ACM.
- Tian, K., Zhang, E., and Shen, X. (2011). A step towards transparent integration of input-consciousness into dynamic program optimizations. In *OOPSLA*, pages 445--462. ACM.
- Wadler, P. (1988). Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231--248.
- Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *TOPLAS*, 13(2).
- Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131--144. ACM.
- Wu, Y. and Larus, J. R. (1994). Static branch frequency and program profile analysis. In *MICRO*. IEEE.