DESPACHO DINÂMICO DE OTIMIZAÇÕES SENSÍVEIS AO CONTEXTO

GABRIEL POESIA REIS E SILVA

DESPACHO DINÂMICO DE OTIMIZAÇÕES SENSÍVEIS AO CONTEXTO

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

Julho de 2017

GABRIEL POESIA REIS E SILVA

DYNAMIC DISPATCH OF CONTEXT-SENSITIVE OPTIMIZATIONS

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

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte

July 2017

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Silva, Gabriel Poesia Reis e

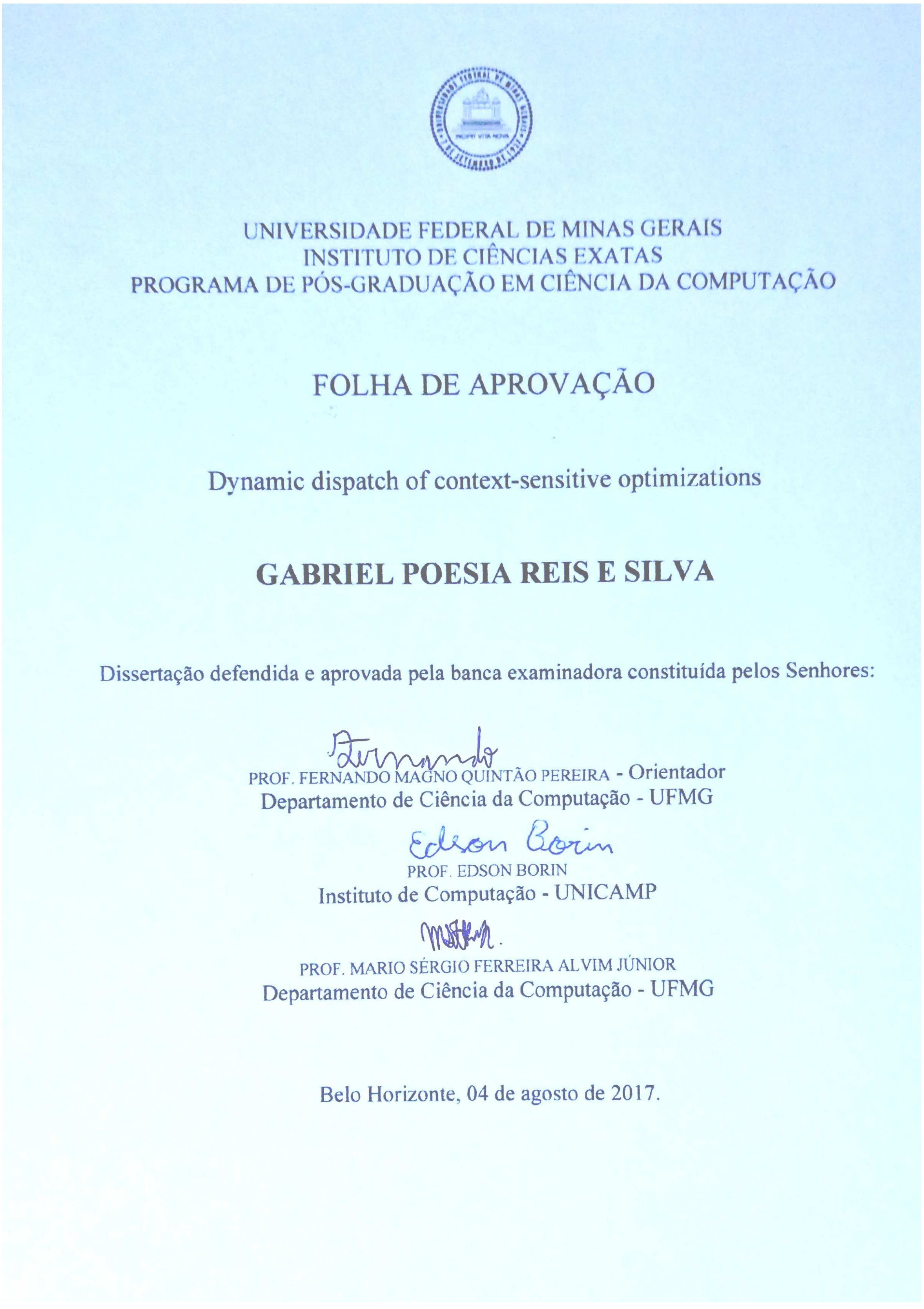
S586d Dynamic dispatch of context-sensitive optimizations / Gabriel Poesia Reis e Silva — Belo Horizonte, 2017.
xxii, 64 p.:il.; 29 cm.

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

Orientador: Fernando Magno Quintão Pereira

1. Computação – Teses. 2. Compiladores (Computadores). 3. Linguagem de Programação (Computadores). 4. Otimização combinatória. I. Orientador. II. Título.

CDU 519.6*33(043)



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

FOLHA DE APROVAÇÃO

Dynamic dispatch of context-sensitive optimizations

GABRIEL POESIA REIS E SILVA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

Edson Borin
PROF. EDSON BORIN
Instituto de Computação - UNICAMP

[illegible]
PROF. MARIO SÉRGIO FERREIRA ALVIM JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 04 de agosto de 2017.

Acknowledgments

O Mestrado constituiu um período de muito aprendizado em minha vida. Agradeço inicialmente aos que fizeram com que essa experiência fosse, antes de mais nada, possível: a todos os funcionários e professores do DCC e da UFMG por sempre prezarem pela excelência, a meus pais pelo suporte indispensável e ao CNPq pelo fomento.

Além da simples possibilidade, agradeço aos que fizeram com que eu pudesse aprender e crescer neste período. Primeiramente, agradeço ao Fernando, meu orientador, pela direção precisa nos dois trabalhos que realizamos, pelo apoio constante em minhas decisões, e por tornar todo o processo divertido e proveitoso. Posso afirmar que minha decisão pela área de pesquisa teve como principal influência a possibilidade de tê-lo como orientador, já que áreas na Ciência da Computação que me interessavam não faltavam. Agradeço também aos colegas de laboratório, pela experiência trocada e por terem desenvolvido ferramentas essenciais para meus projetos. Em especial, agradeço ao Péricles e ao Gleison pelo trabalho no `dawncc`, sem o qual meu primeiro trabalho no Mestrado não existiria, e ao Breno pela colaboração em dois trabalhos: um que teve como fruto esta dissertação e outro um artigo em OOPSLA 2017.

Muitas das ideias que tive no Mestrado não teriam surgido sem a experiência que tive na Maratona de Programação. Assim, agradeço também aos que fizeram parte desse processo intenso e inesquecível de crescimento. Primeiramente, agradeço ao Leo e ao Henrique por tudo o que nos transmitiram – muito mais que novos algoritmos. Agradeço ao Lucas e ao Luiz por termos formado o eterno Starters Edition e a Dilson, Filipe, Guelman e Rennó pela "rivalidade" extremamente proveitosa e divertida. Tudo o que aprendi com todos vocês será dívida eterna.

Por fim, agradeço a todos os demais que fizeram parte especial deste período: Victor e Nildo, companheiros de graduação, de Mestrado e de startups, Alessandra e Wesley, amigos com quem quero caminhar enquanto os pés me permitirem, e Melanie, por ter tornado meus dias mais doces.

"Context and memory play powerful roles in all the truly great meals in one's life."

(Anthony Bourdain)

Resumo

Construir análises sensíveis ao contexto escaláveis é um problema que vem sendo frequentemente trabalhado pela comunidade de compiladores, com sucesso. Porém, a implementação de otimizações sensíveis ao contexto continua sendo desafiadora. O principal problema que desencoraja os compiladores de implementarem tais otimizações é o crescimento no tamanho do código. Com clonagem de funções ou inlining, duas técnicas conhecidas para a implementação de especializações sensíveis ao contexto, o tamanho do código pode crescer exponencialmente no pior caso. Ambas as técnicas são baseadas em criar cópias especializadas do código para cada contexto. Contudo, as duas técnicas precisam criar cópias de todas as funções no caminho de chamadas que leva a cada otimização, ainda que isto envolva copiar funções que não serão otimizadas. Neste trabalho, propomos uma solução para este problema. Utilizando uma combinação de despacho dinâmico e uma máquina de estados para controlar as transições entre os contextos dinamicamente, nosso método implementa otimizações completamente sensíveis ao contexto necessitando apenas copiar as funções que serão otimizadas, mas não o caminho de chamadas até elas. Apresentamos nossa abordagem em Minilog, uma linguagem mínima que possui todos os recursos necessários para aplicar o método proposto, e provamos sua corretude. Implementamos nosso método na infraestrutura de compiladores LLVM, utilizando-o para otimizar programas com propagação de constantes completamente sensível a contexto. Nossos experimentos nos benchmarks do LLVM Test Suite e do SPEC CPU2006 mostram que nosso método escala significativamente melhor em termos de espaço que clonagem de funções, gerando binários em média 2.7x menores, adicionando em média 8.5x menos *bytes* ao implementar a mesma otimização. Os binários gerados utilizando nossa técnica tiveram tempo de execução muito semelhante aos gerados com clonagem tradicional. Além disso, utilizando essa classe de otimizações ainda pouco explorada, conseguimos *speed-ups* de até 20% em alguns benchmarks quando comparados a LLVM -O3.

Keywords: Compiladores Otimizantes, Linguagens de Programação, Otimizações

Sensíveis ao Contexto, Propagação de Constantes.

Abstract

The compilers community has dedicated much time and effort to make context-sensitive analyses scalable, with great profit. However, the implementation of context-sensitive optimizations remains a challenge. The main problem that discourages compilers from making use of such optimizations is code size growth. With either function cloning or inlining, two known techniques for context-sensitive specialization, the code can grow exponentially in the worst case. Both techniques are based on creating copies of all functions in the call path that leads to each optimization, even when that comprises copying functions that are not optimized, in order to keep track of the calling context. In this dissertation, we propose a solution for that problem. Using a combination of dynamic dispatch and a state machine to dynamically control the transitions between calling contexts, our method implements fully context-sensitive optimizations only needing to copy optimized functions. We present our approach in Minilog, a minimal programming language that contains the necessary constructs to apply the proposed method, and prove its correctness. We have implemented our method in the LLVM compiler infrastructure, and used it to optimize programs with fully context-sensitive constant propagation. Our experiments in LLVM Test Suite and SPEC CPU2006 show our method scales significantly better than function cloning in terms of space: it generates binaries that are, on average, 2.7x smaller, adding 8.5x less bytes to implement the same optimizations. In terms of speed, our binaries perform similarly to those generated using function cloning. By using this largely unexplored class of optimizations, we have observed speed-ups of up to 20% in some benchmarks when compared to LLVM -O3.

Palavras-chave: Optimizing Compilers, Programming Languages, Context-Sensitive Optimizations, Constant Propagation.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Context-sensitive compiler optimizations are known to be significantly more precise in various scenarios when compared to their context-insensitive counterparts (Lattner et al. [2007]; Whaley and Lam [2004]). Due to this observation, the academia and the industry have devoted much time and energy to scale up context-sensitive analyses (Arzt et al. [2014]; Das [2003]; Emami et al. [1994]; Fähndrich et al. [2000]; Feng et al. [2014]; Ghiya and Hendren [1996]; Hind et al. [1999]; Might et al. [2010]; Milanova [2007]; Milanova et al. [2014]; Oh et al. [2014]; Wei and Ryder [2015]; Wilson and Lam [1995]). Presently, state-of-the-art context-sensitive analyses can be used in mainstream compilers, as demonstrated by Lattner et al. [2007], or by Li et al. [2013]. Nevertheless, even though we have today the technology to retrieve context-sensitive information from large programs with speed and accuracy, making effective use of this information with the goal of optimizing programs still seems an unsolved problem.

Applying the results of context-sensitive analyses is a challenge. The usual approach towards this end is to perform some optimization once the context-sensitive analysis shows that some code transformation is safe for every possible context. This modus operandi misses one important opportunity for code improvement: *specialization*. If the compiler proves that an optimization is safe for certain contexts and unsafe for others, then it can apply the optimization selectively, in contexts where it is safe to do so. This is accomplished via function inlining or cloning (Das [2003]; Hall [1991]; Metzger and Stroud [1993]; Petrashko et al. [2016]; Cooper et al. [1993]). Nevertheless, due to code size expansion, compilers either do not resort to this kind of specialization or apply it in a very limited way. One of the key challenges preventing specialization is the fact that not only the optimized functions must be replicated, but whole paths of procedures within the program's call graph. Such paths lead from the function where information first becomes available (thus enabling the optimization) to the function that is effectively transformed. Every function called in between must be either cloned or inlined, as we further explain in Section 1.1.

However, the fact that mainstream compilers avoid applying context-sensitive specialization does not mean that such approach is not in demand. For instance, just-in-time compilers often perform context-sensitive specializations. In this case, context-sensitive information is readily available, because the runtime engine has access to the heap state once it moves from interpretation to compilation. Examples of code specialization in the JIT world include constant propagation (Santos et al. [2013]), type speculation (Gal et al. [2009]; Hackett and Guo [2012]), method resolution (Hölzle et al. [1991]) and operation specialization (Wang et al. [2014]). For JIT compilers, code size explosion is not a problem: if a new version of a specialized function needs to be produced, its old binary is usually trashed. Because the discarded function will never be called again, there is no need to keep clones of non-specialized functions just to single out its calling context. Similarly, static compilers perform specialization in restricted scenarios. For instance, GCC, at the -O3 optimization level, creates specialized versions of a function if it can infer that some of its parameters are constant in specific contexts, and it considers the tradeoff between speedup and code size increase to be worthwhile[1]. Similarly, specialization of generic functions is a well-known optimization for languages with parametric polymorphism (Petrashko et al. [2016]). In this paper, we present a technique that makes this sort of context-sensitive specialization practical in general for static compilers.

We tackle the excessive code size growth problem in context-sensitive optimizations by using an implicit state machine to track the calling context, minimizing code duplication. By using our method, the number of function clones that must be created is exactly the number of optimization opportunities found by the context-sensitive analysis, regardless of the length of the call paths that must be taken to reach optimized functions. Notice that the size of our state machine is still directly proportional to the number of contexts in the target program. Thus, it can be exponential in the program size. However, even in this worst-case scenario, it is a data structure that grows, not code, contrary to traditional context-sensitive code specialization, as performed by Hall [1991] or Das [2003], for instance. The generated code then uses dynamic dispatch to decide which version of a function must be called at runtime by querying the state machine. The overhead of such calls and state machine updates only impacts relevant paths in the call graph: regular function calls are performed in paths that do not contain optimized functions. Furthermore, our approach naturally

[1]`https://gcc.gnu.org/svn/gcc/tags/gcc_6_3_0_release/gcc/ipa-cp.c`

handles recursion, making it applicable in situations in which function inlining could not be used to produce specialized code.

To validate our ideas, we have implemented a context-sensitive version of interprocedural constant propagation in LLVM (Lattner and Adve [2004]). Our implementation clones every function that is amenable to code specialization. Typical clone-based optimizations, in contrast, clone every function in contexts leading to specialized procedures. We have tested our technique on 191 benchmarks, taken from the LLVM test suite and SPEC CPU2006. The executables produced with our approach are practically as fast as those produced with full cloning. Programs optimized with our approach, which introduces some runtime overhead, took 1517.4s seconds to run in total, versus 1515.4s seconds taken by the programs transformed with full cloning. This small difference indicates that the overhead imposed by our state machine is minimal. On the other hand, its benefit, in terms of code size reduction, is significant. Our approach greatly outperforms traditional cloning-based specialization: we add 33.37MB to the binaries of the benchmarks in total, whereas full cloning uses 281.94MB of additional space (8.5 times more) to implement the same optimizations. This difference is significant, given that the optimized binaries generated by LLVM without our transformations amount to 107.9MB – thus, cloning-based optimization more than doubled the total size of the binaries, whereas our approach implemented the exact same optimizations with a size overhead of 30.9%.

Section 1.1 begins the discussion with important definitions that will set the ground for the presentation of our work. Section 1.2 contains a brief presentation of the central idea in the dissertation: a technique to implement context-sensitive optimizations which creates a number of function clones proportional to the number of routines that can be optimized, instead of to the size of the context call tree.

1.1 Motivation and preliminary definitions

The vast majority of programming languages provide developers with the abstraction of *functions* (or *procedures*, *subroutines*, *methods*, etc) as an execution unit. Functions can call one another, and even call themselves when recursion is supported. The execution of the program is often supported by a call stack. A central concept for our work is the notion of the *calling context* of an activation of a function f. Basically, the *calling context* is a snapshot of the call stack when execution is at function f. Definition 1.1 formalizes this notion. We assume we can refer to the instructions in a program using some consistent numbering scheme, which is typically trivial to produce (e.g. line numbers if we disallow having two statements on the same line).

Definition 1.1 (Contexts)**.** A Context C of a program P is a sequence of integers that index *call* instructions in P. We denote C by the sequence $C = C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_n$. The empty context, in which execution begins, is denoted by $C_\emptyset$. The function invoked by C_n (the last call instruction of C) is the *active function* of C.

If a dataflow analysis propagates information across the boundaries of functions, then it is called *interprocedural.* Interprocedural analyses can be either *context-insensitive* or *context-sensitive.* This dissertation focuses on the latter. For an informal definition of these families of dataflow analyses, we quote Khedker *et al.*:

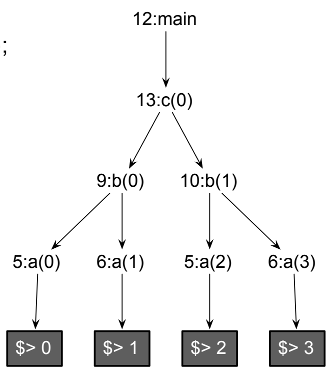
> "*If the information discovered by an interprocedural analysis for a function could vary from one calling context of the function to another, then the analysis is context-sensitive. A context-insensitive analysis does not distinguish between different calling contexts and computes the same information for all calling contexts of a function.*" (Khedker et al. [2009])

Context-sensitive analyses are more precise than their context-insensitive counterparts; however, they are also more expensive. This cost is high because the amount of static information necessary to track different calling contexts can be exponential in the size of non-recursive programs ([Nielson et al., 1999, Sec. 2.5.4]). Moreover, in face of recursion, general context-sensitive analysis is undecidable, as per Reps [2000].

If the problem of statically obtaining context-sensitive information is difficult, the implementation of compiler optimizations that use this information seems to be even harder. The compiler literature describes two main ways to enable context-aware optimizations: inlining (copying the callee's body to the call site) and cloning (creating a separate copy of a function to be called only from some contexts). Many industrial compilers, such as gcc, LLVM, Open64, Jikes and Mozilla's IonMonkey implement inlining at higher optimization levels. Cloning is also found in mainstream products. For instance, gcc may clone a function when doing constant propagation if it judges such optimization will be worthwhile. Moreover, Scala clones generic functions that are marked with the @specialized annotation. The difficulty of applying either inlining or cloning stems from two problems: compilation-time and code-size explosion. The number of calling contexts in a program can be exponential on its size, and it is possible that each such context can be specialized in a different way. As a result, the unrestricted application of any of these techniques might result in an optimized program that is exponentially larger than its original version. Example 1.1 illustrates

```
1  void a(int x) {
2    printf("%d\n", x);
3  }
4  void b(int x) {
5    a(2*x);
6    a(2*x + 1);
7  }
8  void c(int x) {
9    b(2*x);
10   b(2*x + 1);
11 }
12 int main() {
13   c(0);
14 }
```

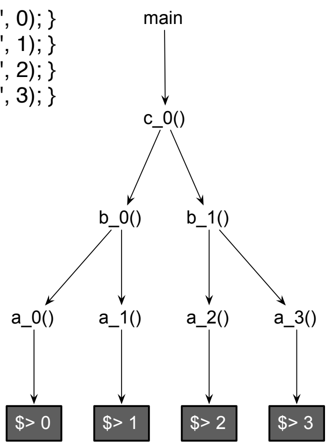
(a)



(b)

```
void a_0() { printf("%d\n", 0); }
void a_1() { printf("%d\n", 1); }
void a_2() { printf("%d\n", 2); }
void a_3() { printf("%d\n", 3); }
void b_0() {
  a_0();
  a_1();
}
void b_1() {
  a_2();
  a_3();
}
void c_0() {
  b_0();
  b_1();
}
int main() { c_0(); }
```

(c)



(d)

Figure 1.1. (a) Program before context-sensitive optimization. (b) Calling context tree of the original program. (c) Program after context-sensitive constant propagation. (d) CCT of the optimized program.

this shortcoming. The example mentions the *Calling Context Tree* (CCT), a graphic representation of every possible calling context in a program (Ausiello et al. [2012]).

Example 1.1. Figure 1.1 (a) shows a C program containing four different invocations of function a. Context-sensitive constant propagation lets us replace each instance of a's argument x, with a constant. The resulting program appears in Figure 1.1 (c), and its calling context tree appears in Figure 1.1 (d). Notice that a context-insensitive analysis would not be able to carry out this transformation, as the value of x in lines 5 and 6 of Figure 1.1 (a) varies during program execution.

To implement interprocedural context-sensitive constant propagation in the program of Figure 1.1 (a), we had to produce a clone of each function, for each calling context where that function can be invoked. This fact is unfortunate, because the actual effect of the optimization, e.g., the replacement of a's argument x by a constant, can only be observed in function a itself. The other clones only exist to distinguish one context from the other. Henceforth, we shall call the optimized function a *leaf*, and the call sites leading to its invocation the *path*. This example takes us to one of the key shortcomings of existing context-sensitive optimizations: the number of clones necessary to implement unrestricted context-sensitive optimizations is proportional to the number of calling contexts, not to the number of instances of functions actually optimized.

This shortcoming has motivated a long string of research to make the implementation of *fully (unrestricted) context-sensitive optimizations* a viable endeavor. Table 1.1

Approach	**CS analysis?**	**CS optimization?**	**Clones needed**
Classical optimizations	No	No	-
Li et al. [2013]	Yes	No	-
Full inlining	Yes	Yes	One per calling context
Hall [1991]	Yes	Yes	One per node in optimization path
This work	Yes	Yes	One per optimized context

Table 1.1. Previous work in perspective, considering whether the analysis and the optimization are context-sensitive (CS).

provides some perspective on previous attempts, contrasting them with this work. The table distinguishes the static analysis that enables an optimization from the optimization itself. The last four techniques rely on fully context-sensitive (CS) static analyses to discover optimization opportunities. However, to avoid cloning too many functions, they resort to different strategies. Li et al. [2013] uses context-sensitive information to optimize functions context-insensitively. There is a large benefit in precision when context is taken into consideration even in this scenario, as the authors show. We shall say, in this case, that the static analysis is context-sensitive, but the optimization is context-insensitive (CI). Such approach seems to be very popular, as we can infer from the Related Work Section of Sridharan and Bodík [2006]. This is in contrast with full inlining, which is the epitome of the context-sensitive optimization. However, full inlining is not practical, due to code-size explosion. To deal with this problem, Hall [1991] only clones paths that are bound to different static facts inferred by the context-sensitive analysis. If every path leads to a different version of a leaf, then this approach degenerates to full inlining. Finally, as we clarify in Section 1.2, our approach only clones leaves. We might transform functions in the calling path; nevertheless, we always end up with a single implementation of them.

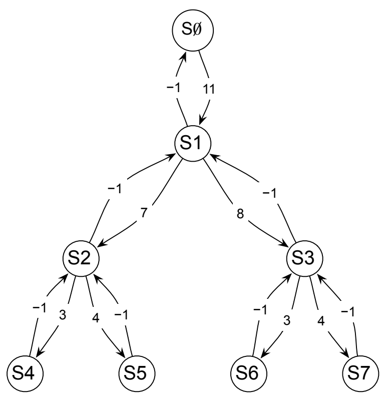
1.2 State Machines to the Rescue

To circumvent the code-size explosion problem, we implement context-sensitive optimizations via a combination of a state machine and dynamic dispatch. The guarantee that this arrangement provides is stated in Definition 1.2, and illustrated in Example 1.2.

Definition 1.2 (Guarantee)**.** Let A be a context-sensitive analysis, and O be a context-sensitive optimization. If P is a program, then, for every function $F \in P$, we let $A(F, C)$ be the facts that A infers about F at context C. We say that F is a *leaf* if $A(F, C)$ is non-trivial. Non-trivial static facts about a context C allow O to specialize F in C, i.e., $O(F, C) \neq= F$. Hence, leaf functions are amenable to optimization in some

```
void a_0(int x) { printf(“%d\n”, 0); }
void a_1(int x) { printf(“%d\n”, 1); }
void a_2(int x) { printf(“%d\n”, 2); }
void a_3(int x) { printf(“%d\n”, 3); }
void b(int x) {
  { auto p = transition(3); p(x);
  transition(-1); }
  { auto p = transition(4); p(x);
  transition(-1); }
}
void c(int x) {
  { auto p = transition(7); b(x);
  transition(-1); }
  { auto p = transition(8); b(x);
  transition(-1); }
}
int main() {
  { auto p = transition(11); c(x);
    transition(-1);}
}
```

(a)



(b)

```
void* transition(int next) {
  switch(state) {
    case 2: {
      switch(next) {
        case 3:
          state = 4;
          return &a_0;
        case 4:
          state = 5;
          return &a_1;
        case -1:
          state = 1;
          return NULL;
        default: error();
      }
      ...
    }
  }
}
```

(c)

Figure 1.2. (a) Program after context-sensitive optimization. (b) State machine that tracks context changes. (c) The transition function, which implements state switching and selects which version of an optimized function to call.

contexts. To implement O, we shall produce one clone of F for each $A(F, C)$ that yields non-trivial static facts. We shall not create clones for contexts for which A provides only trivial facts.

Example 1.2. Figure 1.2 (a) shows an optimized version of the program earlier seen in Figure 1.1. The context-sensitive optimization is constant propagation. The calling context tree has four leaves – the four different activations of function a – each one receiving a different constant as argument. The program in Figure 1.2 (a) contains calls to a function transition, which is in charge of controlling a state machine that tracks calling contexts. Such state machine is shown in Figure 1.2 (b). At each state switch, transition might return a pointer to the next function to be invoked. An example of the code that performs this action appears in Figure 1.2 (c). We are showing the implementation of state S2, that decides, among two optimized versions of function a, (a_0 and a_1), which one should be invoked next.

The state machine lets us distinguish, at run-time, the different contexts that we have analyzed statically. Before invoking a function f, we switch the program state, passing appropriate selectors to the state machine. A new state change happens also when f returns. Notice that function invocation is the sole event that determines a change in context. When necessary to invoke a function g that we have been able to optimize, the current state lets us determine which version of g should be called. To implement this selection, we represent g as a pointer, whose target is defined by the state machine. In other words, we are creating a *virtual table* that contains entries for every function that could be optimized due to context-sensitive information produced statically. Introducing indirect function calls makes it harder for other analyses to reason about call targets; however, in our implementation, we run our algorithm late in the compiler's pipeline of transformations, so that optimizations that depend on such information have already taken place.

In Figure 1.2 (a) we have four clones of function a, the only routine that could be transformed by our initial example of optimization: constant propagation. The other functions, which do not present optimization opportunities, remain unchanged. By implementing context-sensitive optimizations with a state machine, we ensure that the number of clones is upper bounded by the number of disjoint code transformations that can be performed. As we show in Section 4, in practice this number is much smaller than the number of possible contexts that a static analysis must recognize. However, we are not reducing the worst case bound on the number of clones: it is still exponential in the program size. To see how this worst case emerges, it suffices to consider a calling context tree in which every leaf represents a function amenable to a different code transformation.

Chapter 2 puts our work in context with a review of previous related research found in the literature, especially on the topics of *context-sensitive analyses* and *code specialization techniques*. Chapter 3 provides a formal description of our approach, including the semantics of MiniLog, a programming language with a minimum set of constructs necessary to implement our method. It also contains correctness proofs, showing that our state machines are equivalent to full code specialization. We have implemented MiniLog in Prolog, so that one can validate our formal notation in an actual interpreter. Chapter 4 describes the implementation of our technique in LLVM, together with its empirical evaluation. To the best of our knowledge, our tool brings in the first implementation of a fully context-sensitive code specialization strategy that does not resort to code replication to track calling contexts. We have chosen constant propagation as our example of context-sensitive code specialization because it is easy to implement, and extensive: it is hard to think about a static optimization that gives origin to a larger number of specialized functions. Finally, Chapter 5 concludes with final remarks and directions for future work. This dissertation is concerned about the following hypothesis:

> *Our research thesis:* It is possible to make context-sensitive optimizations practical by minimizing the code duplication currently necessary to implement them.

Chapter 2

Literature Review

This work touches three important subjects within compilers research: (a) the implementation of context-sensitive analyses, (b) the specialization of program code (context-sensitive or insensitive) and (c) the representation and tracking of calling contexts. Most of the previous works we found concern the use of *context-sensitive analyses* in *context-insensitive optimizations*. In Section 2.1, we look in depth at context-sensitive analyses, which brings into perspective different forms of representing calling contexts and of making context-sensitivity scale to large programs. We also note that not all such analyses can produce the information necessary for implementing the kind of optimization we use in our work. Having gone through analyses, in Section 2.2 we the specialization of program code for different calling contexts. Finally, our method depends on a representation of calling contexts and the maintenance of the current calling context during run-time. We have employed a DFA for this task. We compare this choice with previous techniques in Section 2.3.

Even though these are well-known topics, being even discussed in general compiler textbooks, we believe that our contribution is unique when compared to previous art. To the best of our knowledge, there are no implementations of fully context-sensitive optimizations; at least, not if we consider optimizations that can be widely applied – as constant propagation, for instance. Nevertheless, mainstream compilers and research artifacts do use context-sensitive information towards program improvement. To this end, they often rely on some compromise between precision and applicability. In this chapter, we discuss some of these compromises.

2.1 Context-Sensitive Analyses

This dissertation provides a new method for implementing context-sensitive optimizations. Our method is analysis-agnostic: it assumes that context-sensitive analyses have run and decided which optimizations should be applied in which calling contexts. This section sheds more light on how such analyses have been developed in the literature. It also helps put our method into perspective, since the input we assume in our algorithm (optimizations for specific calling contexts) does not match the output of many of the proposed analyses, which produce summaries for the information gathered about functions when all calling contexts are aggregated. Rather, we expect the analyses to produce specializations for specific calling contexts.

A vast body of previous work is concerned with answering queries related to the program states that are possible in a certain procedure, either in a specific calling context (context-sensitive analysis) or in any of them (context-insensitive). One important classical example is the problem of pointer analysis, which is still the subject of much of the current research in compilers. Pointer analysis consists of statically determining which heap objects can a pointer variable point to in run-time. Particularly important is being able to tell whether it is possible for two pointers to alias – i.e. to point to the same object in run-time. Such information is important because some optimizations can only be applied when the compiler can prove that pointers do not alias. Examples of such optimizations are automatic parallelization and some cases of common subexpression elimination and partial redundancy elimination. The work of Alves et al. [2015] showcases the practical usefulness of that information as an enabler of various compiler optimizations.

One of the most basic representations of contexts used in context-sensitive analyses and optimizations was introduced by Emami et al. [1994]. The authors propose the use of the invocation graph, which they define as a graph where nodes are calling contexts and edges represent function calls that take the program from one calling context to another. They also statically determine which procedures each function pointer can point to, in order to obtain a more precise invocation graph. Invocation graphs allow the analysis to completely avoid summarizing information related to a procedure among its various calling contexts. Indeed, the proposed analysis is very precise. It computes points-to sets for each variable (a set of objects to which each pointer can point to at run-time). The best possible size for a points-to set is 1, i.e. the analysis statically discovers the only object that a pointer may reference. When considering all benchmarks, the proposed analysis computed points-to sets that have an average size of 1.13 – quite close to one. Such precision comes at a cost: the size of the invocation graph can be exponential in the depth of the call graph. Thus, in order to scale to large programs, context-sensitive analyses usually resort to some sort of summarization, without falling back to being interprocedural but context-insensitive (i.e. not differentiating calling contexts at all). Emami *et al.* manage to run their analysis on programs of up to 3,000 lines of C code.

A widely cited example of context-sensitive summarization was proposed by Wilson and Lam [1995]. This work uses interprocedural *transfer functions* for computing points-to sets. However, for a given procedure, the proposed transfer functions do not combine information coming from all calling contexts. Rather, the authors introduce the notion of *partial transfer functions* (PTFs), that merge information from all contexts that have the same points-to sets for their parameters. The use of PTFs relies on the fact that points-to sets tend to repeat between most contexts in pointer analysis problems. Therefore, while the number of different contexts can be very large, it happens in practice that many contexts are associated with the same information by the analysis. Thus, it can scale by not needing to compute information for every distinct context, but only for every distinct realizable parameter points-to sets. This approach was shown to scale better than that of Emami et al. [1994], as Wilson and Lam [1995] runs their implementation on C programs with tens of thousands of lines of code, having on the order of 10^8 distinct contexts.

One disadvantage of both cited approaches is that they need to explicitly represent either each context (Emami et al. [1994]) or each class of contexts with the same points-to information (Wilson and Lam [1995]). While this can initially seem as an unavoidable burden of context-sensitive analyses, the scalability of points-to analysis was taken to a new level by representing contexts in a remarkable way. Whaley and Lam [2004] show how to represent context-sensitive points-to analysis using Datalog, a logic programming language very similar to Prolog. They develop Datalog programs that take input relations describing simple context-insensitive static facts (such as "v is assigned a newly allocated heap object", and "v is passed as parameter p of function f"). The program uses logic programming rules to deduce points-to sets in a context-sensitive way using these facts. An example of such Datalog rule in simple English would be: "If v has points-to set S_1 in context C_1 and v is passed to parameter p from function f called from C_1, then the points-to set of parameter p in context $C_2 = C_1 \to f$ includes the points-to set of variable v in context C_1". The main algorithm consists of processing such rules iteratively until no new information can be derived. The advantage of this approach is that Datalog programs can be translated to operations from relational algebra, which operate on entire relations, not individual tuples. Moreover, relational algebra operations can be efficiently translated into operations on Binary Decision Diagrams (BDDs), a compact data structure for representing binary functions. The authors use both facts in order to compute points-to sets for all distinct contexts (ignoring recursion) without explicitly even listing all of them, but instead by operating on BDDs. After that, queries for explicit contexts can be answered efficiently. BDDs implicitly use the commonalities in the representation of contexts for manipulating large numbers of them at once, instead of individually. This approach has made fully context-sensitive points-to analysis possible on programs significantly larger than previous works, as the authors successfully run their implementation, named `bddbddb` on Java programs with up to 10^{14} distinct contexts, with their analyses running in less than a minute for large benchmarks. Zhu and Calman [2004] and Lhoták and Hendren [2006] are two other works that explore the usage of BDDs for representing calling contexts.

The works of Emami et al. [1994], Wilson and Lam [1995] and Whaley and Lam [2004] are examples of *inclusion-based* points-to analyses. This is one general method of handling context-sensitive information, in which each context-sensitive entity (pointer and heap object, in the case of points-to analyses) has its own associated information set, and relations among entities in different contexts are derived from the program and the call graph. While very precise, it is still hard to scale such analyses to the requirements of industrial compilers. Even `bddbddb` can take almost one minute for large benchmarks, which is suitable from a research perspective but still more than what compilers such as GCC or LLVM are willing to tolerate. An alternative to inclusion-based analyses are the *unification-based* approaches. These analyses collapse different contexts into one single information set when it makes sense for the particular application they are used for. For example, in pointer analyses, the common practice for unification-based algorithms is to derive that the points-to sets of variables a and b are equal when an assignment such as "a = b" is found in the program. Thus, henceforth it would treat a and b as the same variable (i.e. it would *unify* a and b), and any new object either a or b are discovered to point to would figure in the points-to set associated with both variables. In contrast, an inclusion-based analysis would only derive the fact that the points-to set of b is a subset of the points-to set of a (since a can receive any value that comes from b, but the same is not true since b is not assigned the value of a). Inclusion-based analyses are thus more precise, but also more expensive.

Unification-based context-sensitivity, on the other hand, has been shown to scale to large programs under the time and memory constraints industrial compilers have to meet. One of the most notable examples (for being included in LLVM) is the work of Lattner et al. [2007]. The authors present a context and field-sensitive[1] unification-based pointer analysis that was able to scale to very large programs, such as the Linux kernel, in under 3 seconds – a fraction of the time GCC takes to compile this program (with 355K lines of code). Their algorithm is based on first building a local graph for each function that represents intraprocedural points-to relations. Then, two interprocedural passes take place in sequence. First, a bottom-up pass takes the graphs from callers and merges them into callees. Then, a top-down pass merges the graphs from callees into callers. The analysis is then done, having as a result still one graph for each procedure, but one that takes into account the full depth of calls that are made from the function onwards. While their approach (called Data Structure Analysis) is highly scalable, note that the term *context-sensitive* has different implications in Lattner et al. [2007] when compared to Emami et al. [1994] or Whaley and Lam [2004]. Data Structure Analysis is not able to answer queries as precisely as `bddbddb`. In particular, it cannot reason about arbitrary calling contexts. Rather, for one given function, it will unify the points-to information from all calling paths that reach that function, combine that with information from each callee and to the local points-to relations, generating a procedure-level summary. However, context is still taken into consideration since each call site causes the graphs of caller and callee to be merged in a different way. This approach has shown to be precise when compared to other context-insensitive pointer analyses. The authors do not compare to context-sensitive alternatives, likely because an LLVM implementation of them is not available.

Unification and *inclusion* are not the only class of approaches for context-sensitivity found in the literature. A third approach is based on graph reachability, first presented by Reps et al. [1995]. Rather than pre-computing points-to sets for each variable, queries are answered precisely and on-the-fly in polynomial time. Basically, graph reachability-based algorithms label the edges of the call graph with call site identifiers. When given a query such as "is the value of pointer a ever assigned to b?", the algorithm will find a path on the labeled call graph from "a" to "b" that has matching "call" and "return" labels. For example, if function f calls function g in a given line L of the program, the call graph will have an edge from f to g with label L. A valid path from a to b that answers the given query must then traverse the edge labeled L twice: one for the call, and one for the corresponding return. A naïve algorithm that does not consider such constraint could possibly find a path that traverses that edge only once (for the call), then traverses an edge to another function h that is called from g, and then directly returns to f through an $f - -h$ edge. Such path, however, would

[1]A field-sensitive analysis differentiates between different fields of the same structure or object. Thus, if an object has two pointer fields, they may have distinct points-to sets associated with them.

be impossible in an actual program execution , since calls and returns to not match. On the other hand, a proper algorithm restricts its search to paths that belong to a certain *context-free language* (CFL) that matches call and returns labels. Thus, such analyses are usually known as CFL-based.

Using a lossless reduction to a graph reachability problem, Reps [2000] goes further to show that in general, data dependence analysis is undecidable. That means it is impossible to have a 100% precise, deterministic algorithm that is able to tell whether the value of one program variable depends, directly or indirectly, on the value of another variable. While some approaches claim to have an exact and efficient algorithm for data dependence testing, they all refer to the context of a single procedure, when considered out of context Maydan et al. [1991]. Thus, they are not context-sensitive, and the undecidability result of Reps [2000] does not apply to them.

Li et al. [2013] is an example of a recent application of a CFL-based algorithm to pointer analysis. The authors propose a context- and field-sensitive modeling based on the Value Flow Graph (VFG), a data structure they introduce. The VFG relates local and global variables, as well as structure fields, using assignment statements found in the program. Besides these intraprocedural relations, interprocedural edges from formal parameters and actual parameters are derived from function call sites. Their algorithm clones the VFGs for different contexts when the information at each context may be computed differently. On the other hand, when it can prove the information in two contexts will always be the same, they will share their VFGs. Their approach is as precise as inlining all function calls. Key to their scalability is a heuristic they use to limit the number of clones created: their set a hard limit on how many clones per calling context can be generated. With this cap, their implementation scales to all SPEC CPU2006 benchmarks; however, without it, it cannot run on the largest benchmarks. Even with this limit, this CFL-based approach is significantly more precise than Data Structure Analysis (up to 20x in some cases). Still, it takes a few minutes to run and/or uses more than 1GB of memory in the largest programs in CPU2006, which is considered unsuitable for adoption by traditional industrial compilers.

All these analyses can be used to provide information for later optimizations. In many cases, context-sensitivity is used to improve the precision of the information that *context-insensitive* optimizations have access to. For example, the points-to analysis of Lattner et al. [2007] ends up computing a single unified graph for each function. This graph summarizes information taken from all calling contexts that reach that function. However, because it is constructed taking callers and callees into consideration, it is significantly more precise than what an equivalent graph constructed using only locally available information would provide. Also, it is more powerful than interprocedural context-insensitive analyses since call paths are taken into account, even though in a limited way because unification takes place. However, because the output of such analysis is function-level information, even if the analysis in Lattner et al. [2007] is context-sensitive, it is not suitable for (or aimed at) implementing a context-sensitive optimization. In this dissertation, we present a method for implementing context-sensitive optimizations of a form that assumes that different calling contexts should be optimized differently. It is obviously not necessary that all distinct contexts are optimized in order to use the method we present in Chapter 3, since the number of contexts can be exponential in program size and that would mean we cannot inherently scale to even moderate-sized programs. However, our method assumes that distinct contexts have different information associated with them, raising the possibility of specializing the function for some contexts based on facts discovered by the analyses. Thus, some of the presented analyses are suitable to feed our method, and some are not.

In particular, we first represent contexts and optimized contexts with a structure that is similar to Emami's Invocation Graph (Emami et al. [1994]). The Optimization Tree we define in Chapter 3 is a subgraph of the Invocation Graph. We then use a finite state machine to represent transitions between calling contexts, similar to Reps [1997, 2000].

As a last note on terminology, many analyses use the term *heap cloning*, or just *cloning* to refer to the fact that heap objects from different calling contexts have different information sets associated with them. For example, in Lattner et al. [2007] and Li et al. [2013], if two local variables are assigned the return value of two invocations of some function f, and f returns the result of calling *malloc*() (i.e. a newly allocated memory region), such variables will be pointed to two different heap objects, since the algorithms differentiate between where function f was called in the code. Thus, these analyses are said to *clone* the representation of the object returned by f when f is reached from different paths. However, that is different from *cloning* in the realm of code specialization, which we discuss next, in which the term refers to the creation of copies of a procedure's code that may be specialized differently, and that will be present in the final generated binary.

2.2 Specialization of program code

One way of enabling aggressive compiler optimizations is to specialize program code for specific cases. Function cloning is a well-known way to enable code specialization. Compiler textbooks, namely Kennedy's [Kennedy and Allen, 2002, pp.594] and Grune's [Grune et al., 2012, pp.325], contain brief descriptions of this technique. We recognize three different ways to use function cloning towards the production of specialized code: static, dynamic and hybrid. These approaches depend on how clones are produced, and when particular versions are chosen. This might happen, for instance, if the compiler discovers some information that holds in a specific calling context (e.g. some function parameter is constant known at compile time) but not in all others; then, it can produce a special version of the active function to be called only in that calling context (e.g. a clone in which the parameter has been replaced by its known constant value). This is what we call *static specialization* and discuss in Section 2.2.1. Another possibility is that the compiler generates specialized code that is optimized under a given assumption, and then checks for that assumption during run-time before running into the specialized code. We call this *hybrid specialization*, since it involves run-time decisions and static code generation. We review hybrid techniques in Section 2.2.2. Finally, just-in-time compilers perform code specialization during running time, after observing the program's behavior. We review such approach, *dynamic specialization*, in Section 2.2.3.

2.2.1 Static Specialization

Static code specialization is the oldest and most well-known technique, among the three categories that we have listed. Two techniques have been employed to specialize code in order to take advantage of information available only in some calling contexts: *inlining* and *cloning*. Both methods are based on creating specialized copies of a function's code. Inlining consists of replacing a function call by a copy of the body of the called function. After inlining has been performed, the compiler can specialize the inlined procedure using locally-available facts. Very similarly, cloning involves the creation of a copy of a function that is only called in contexts that share certain interprocedural information. For example, if a given function f takes two integer parameters x and y, and in many call sites the constant value 1 is passed as the value of x, the compiler may create a clone of f where that parameter is replaced by a constant, and further optimizations (such as constant propagation) can then take place. The advantage of cloning over inlining is that different calling contexts can call the same clone when the information available in them is equivalent for the purposes of optimization. Inlining, on the other hand, will always create a distinct copy of the code for each call site in which the procedure is inlined. On its positive side, inlining removes the function call overhead, while cloning does not.

The most extensive discussion about inlining and clone-based optimizations that we are aware of can be found in Mary Hall's Ph.D. dissertation (Hall [1991]). Hall shows that unrestricted inlining and cloning are usually detrimental to program performance, even when they enable many optimization opportunities. One key factor is that programs tend to get significantly larger when aggressive cloning and/or inlining are used, which in turn worsens the locality of reference of the binary, and instruction cache misses can hurt performance significantly. Based on this observation, Hall developed an interprocedural constant propagation algorithm that performs cloning only when the target calling context is likely to have a large pay-off optimization from the parameters to be turned into constants. She managed to get significant performance improvements while minimizing code size growth. Because of interactions with constant propagation, which we have also observed in our experiments, cloning and inlining decreased the final binary size in some benchmarks in cases in which they allowed aggressive dead code elimination.

Nowadays, compilers seldom use cloning, while restricted versions of inlining are common, especially combined with cheap heuristics (Cavazos and O'Boyle [2005]). Cloning is the default choice to produce specialized versions of functions that use parametric polymorphism. Stucki and Ureche [2013] have shown that performance improvements of up to 30x are possible when generic functions are specialized to particular types. Other examples along similar direction include Dragos and Odersky [2009], Sallenave and Ducournau [2012], and Petrashko et al. [2016].

2.2.2 Hybrid Specialization

Hybrid specialization combines static and dynamic information to customize program parts to certain inputs or certain events. For instance, Samadi et al. [2012] and Tian et al. [2011] generate programs containing distinct routines to handle different kinds of inputs. Run-time checks are then employed to select which version of a procedure should be called before its invocation. Another example of hybrid specialization based on cloning concerns the line of work known as *Run-time Pointer Disambiguation* (Alves et al. [2015]; Sperle Campos et al. [2016]; Rus et al. [2002]). This technique consists of producing run-time checks that, if satisfied, are enough to prove the absence of aliasing between pointers. Such guards might lead to specialized versions of functions, which have been compiled under the assumption that pointer arguments cannot alias. These optimized functions can be often amenable to vectorization and enable opportunities for classical optimizations such as common sub-expression elimination, which can benefit from aliasing information. The authors achieve performance gains of up to 8.7% on top of LLVM -O3 on the Polybench benchmark suite.

The methods presented in this dissertation fall into the hybrid category. We statically generate a state machine with information about which optimizations can be applied in relevant calling contexts. This state machine is then updated and queried during program executed. Thus, we also combine static and dynamic information. We do so with a novel purpose: we have not found any previous work which uses context tracking in order to select program optimizations during run-time.

2.2.3 Fully Dynamic Function Specialization

All the code specialization techniques seen previously are static: information is collected statically by the compiler, and then used to produce customized versions of functions. When the selection of statically-generated function versions is performed differentiates static and hybrid approaches. On the other end of the spectrum, fully dynamic code specialization is the norm in just-in-time compilers (JITs) (Gal et al. [2009]; Hölzle et al. [1991]; Hackett and Guo [2012]; Santos et al. [2013]). For instance, Santos *et al.* (Santos et al. [2013]) have proposed to generate specialized routines based on the run-time value of the arguments passed to JavaScript functions. Since compilation happens during run-time, the JIT can observe the actual values of parameters that are frequently passed to functions and specialize based on this information, which is unavailable at compile time.

Dynamic specialization is key in achieving good performance in dynamically-typed languages, since the fact that types are not known statically hinders most traditional compiler optimizations. One example is the work of Ahn et al. [2014]. The authors propose to compile JavaScript functions to machine code assuming the parameters have the types they observed in the first calls to the function, if variations in such types do not happen. The generated binary code includes run-time checks that assert the run-time types match the types assumed during compilation; if such checks fail, execution falls back to the interpreter, otherwise the specialized function is executed. This technique enabled an average performance gain of 36% on a benchmark suite formed of real Web sites, on top of Chrome's V8 compiler (which today implements this optimization).

2.3 Calling Context Representation and Tracking

The methods for implementing context-sensitive optimizations we introduce in Chapter 3 depend on a statically computed finite machine for representing calling contexts and transitions caused by function calls. This DFA is then updated at run-time to keep track of which calling context is currently active at any given moment. Queries to the state machine decide which version of a specialized function must be called in the current context. In this section, we compare this approach to other methods found in the literature.

Context-sensitive specialization has traditionally been performed using either function cloning or inlining (Hall [1991]). Both approaches share two key characteristics. First, copies of the code are used to keep track of calling contexts: the code is modified such that a given section is only executed in one calling context. This makes context-tracking totally implicit at running time, incurring no overhead. However, the code size may grow exponentially. Hall [1991] shows that one must be careful when using either method with the goal of optimizing programs, since the benefit of producing a specialized function may not pay off given the running time cost of growing binary size and hurting spatial locality.

Clients of context-sensitive information also include dynamic bug detection analyses. Tools such as data race or memory leak detectors commonly need to associate heap objects with calling contexts. For instance, when a staleness-based memory leak detector marks an object as stale (i.e. it has not been accessed for long), it usually presents to the developer the calling context in which the object was allocated. In order to accomplish this, the tool needs to dynamically maintain a data structure which represents calling contexts. A traditional choice has been the Calling Context Tree (CCT), in which nodes represent contexts and a callee context is a child of its caller (Sarimbekov et al. [2011]; Spivey [2004]). On every function call and return, updates to CCT nodes are performed. A representation that has been shown to outperform the Calling Context Tree is the Calling Context Uptree (CCU), in which node updates are unneeded (Huang and Bond [2013]). In CCU, the edges go from the callee to the caller; thus, the caller's node is not updated when a function is called. These techniques, however, are too expensive to be used for the implementation of context-sensitive optimizations. They require memory allocations every time a function is called, and even the CCU incurs an overhead ranging from 28 to 67% on instrumented programs. However, it is also employed in a class of problems which has more requirements than context-sensitive specialization. For instance, since it cannot predict when the client analysis will request calling contexts, it must track all context transitions. In our case, since we know beforehand which contexts are optimized, we can restrict tracking only to those calling contexts which can eventually reach an optimized context, adding no overhead to the execution outside that scope.

Another important class of clients of calling context information is profiling tools, since full calling contexts are more useful than static source code locations. Ball and Larus [1996] proposed a scheme for numbering paths inside a function such that it is efficient to keep track of which control-flow path a function executed with minimal instrumentation. Bond and McKinley [2007] builds on the same idea to attribute numbers to calling contexts, maintaining dynamic context information with only a 3% overhead on average. These approaches are, however, probabilistic. For instance, in Bond's numbering scheme, there may be collisions in the computed context number during run-time in such a way that the calling context becomes ambiguous if we only consider the current context's number. In profiling, that issue is irrelevant as long as the probability of collisions does not impact the quality of the sampling. When implementing optimizations, however, any error is intolerable: optimizations cannot afford to change the program's semantics independently of how infrequently they do so.

Chapter 3

Implementation of Context-Sensitive Optimizations

This chapter explains our method for implementing context-sensitive optimizations. In Section 3.1 we introduce MiniLog, a core programming language that gives us the necessary syntax and semantics to explain our ideas. We have implemented MiniLog in Prolog, and have used this implementation to design the algorithms that we present here. Our code transformation engine is parameterized by a context-sensitive optimization, a notion formally defined in Section 3.2. Given such an optimization, Algorithm 2 converts a program P_{orig} into an optimized program P_{min}. This new program relies on a finite state machine to decide which functions are called at each invocation site. The generation of this DFA is the subject of Section 3.5. Section 3.6 states a few properties of our approach, including its correctness guarantees. Section 3.7 describes an alternative, equivalent program transformation algorithm that uses the same underlying DFA, but implements it differently, using slightly more space but generating faster code. Finally, Section 3.8 provides details of our implementation in LLVM.

3.1 MiniLog: Syntax and Semantics

Figure 3.2 describes the syntax of MiniLog. MiniLog has statically-scoped local variables, simple arithmetic and boolean expressions, first-class functions, and conditionals. The only type is integer. The only control flow syntax consists of if-then-else blocks. Iteration is achieved through recursive calls. Notice that the need for recursion, combined with static scoping, requires us special syntax to implement recursive functions, similarly to the fun keyword of ML, or the letrec keyword of Scheme. We shall omit this special syntax from our presentation, in order to keep MiniLog's definition simple.

(a)

```
1  function(a, x,
2    print(x);
3  );
4
5  function(b, x,
6    call(a, 2*x);
7    call(a, 2*x + 1);
8  );
9
10 function(c, x,
11   call(b, 2*x);
12   call(b, input);
13 );
14
15 call(c, 0);
```

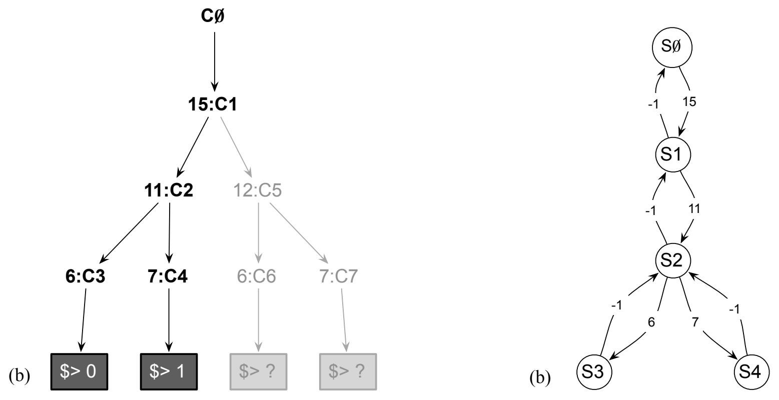


Figure 3.1. (a) Modified version of program seen in Figure 1.1. The unknown variable input, in line 12, prevents constant propagation from optimizing the whole context tree. (b) Calling Context Tree of the program. We have highlighted the Optimization Tree (see Definition 3.3). (c) State Machine produced for this example. Each context in the optimization tree creates a new state.

Nevertheless, we emphasize that MiniLog's actual implementation, as well as all the developments in the rest of our work, handle recursive functions. In Section 3.2 we shall return to some of this syntax, the *forward declaration*, which is necessary for the implementation of our state machines.

Example 3.1. Figure 3.1 (a) shows our original example, introduced in Figure 1.1 (b), implemented in MiniLog. The expression 2*x+1 (originally in line 12 of Figure 1.1 (b)) was replaced by input, to prevent constant propagation. This modification will let us explain how we combine optimized and non-optimized program parts. Notice that Figure 3.2 does not define the syntax of logic and arithmetic expressions, for the sake of space. Nevertheless, we shall assume that such expressions exist, and shall use them freely, as we do in lines 6, 7, and 11 of Figure 3.1 (a).

Figure 3.3 presents the operational semantics of MiniLog. The state of a MiniLog program is given by an *environment* (`Env`: $V \mapsto \mathbb{N}$) and a *store* (`Sto`: $\mathbb{N} \mapsto \mathbb{N}$). The environment maps names to integer numbers representing memory addresses. The store maps these addresses to values. We opted to separate environment from store to be able to emulate pointers in MiniLog. The result of evaluating a program, under some environment and store, is a triple formed by a new environment, a new store, and an *output*. The output is a list of integers, which are produced during the execution of a program. We represent lists using Prolog's syntax, e.g., $[3, 1, 4]$. The output list is the only visible outcome produced by the execution of a program. Thus, two programs, P_1 and P_2 are said to be equivalent if, given the same pair environment/store, they always produce the same output.

Values: $$\frac{V \in \{\ldots, -1, 0, 1, 2, \ldots\}}{\texttt{exp}(V)}$$

Names: $$\frac{A \in (a \ldots z A \ldots Z)*}{\texttt{name}(A)}$$

Variables: $$\frac{\texttt{name}(A)}{\texttt{exp}(A)}$$

Instructions: $$\frac{\texttt{instruction}(I)}{\texttt{program}(I;)}$$

Programs: $$\frac{\texttt{instruction}(I) \qquad \texttt{program}(P)}{\texttt{program}(I; P)}$$

Output: $$\frac{\texttt{exp}(E)}{\texttt{instruction}(\texttt{print}(E))}$$

Arithmetics: $$\frac{\texttt{exp}(E_1) \qquad \texttt{exp}(E_2)}{\texttt{exp}(E_1 \oplus E_2), \oplus \in \{+, -, \times, \text{etc}\}}$$

Allocation: $$\frac{\texttt{name}(A) \qquad \texttt{exp}(E)}{\texttt{instruction}(\texttt{set}(A, E))}$$

Invocation: $$\frac{\texttt{name}(Name) \qquad \texttt{name}(Arg) \qquad \texttt{exp}(Ret)}{\texttt{instruction}(\texttt{call}(Name, Arg, Ret))}$$

Assignment: $$\frac{\texttt{name}(A) \qquad \texttt{exp}(E)}{\texttt{instruction}(\texttt{def}(A, E))}$$

Function: $$\frac{\texttt{name}(Name) \qquad \texttt{name}(Arg) \qquad \texttt{program}(Body)}{\texttt{instruction}(\texttt{function}(Name, Arg, Body))}$$

Return: $$\frac{\texttt{exp}(E)}{\texttt{instruction}(\texttt{return}(E))}$$

Conditional: $$\frac{\texttt{exp}(E) \qquad \texttt{program}(P_1) \qquad \texttt{program}(P_2)}{\texttt{instruction}(\texttt{ifelse}(E, P_1, P_2))}$$

Figure 3.2. The Syntax of MiniLog.

$$\texttt{[chain]:} \quad \frac{\begin{array}{c}\langle P_1, \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}', \texttt{Sto}', \mathcal{O}_1\big) \\ \big\langle P_2, \texttt{Env}', \texttt{Sto}'\big\rangle \to \big(\texttt{Env}'', \texttt{Sto}'', \mathcal{O}_2\big) \qquad \mathcal{O} = \texttt{append}(\mathcal{O}_1, \mathcal{O}_2)\end{array}}{\langle P_1; P_2, \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}'', \texttt{Sto}'', \mathcal{O}\big)}$$

$$\texttt{[def]:} \quad \frac{\mathcal{E}(E, \texttt{Env}, \texttt{Sto}) = V \qquad (\texttt{Env}', \texttt{Sto}') = DefVar(N, V, \texttt{Env}, \texttt{Sto})}{\langle \texttt{def}(N, E), \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}', \texttt{Sto}', []\big)}$$

$$\texttt{[set]:} \quad \frac{\mathcal{E}(E, \texttt{Env}, \texttt{Sto}) = V \qquad \texttt{Sto}' = SetVar(N, V, \texttt{Env}, \texttt{Sto})}{\langle \texttt{set}(N, E), \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}, \texttt{Sto}', []\big)}$$

$$\texttt{[print]:} \quad \frac{\mathcal{E}(E, \texttt{Env}, \texttt{Sto}) = V}{\langle \texttt{print}(E), \texttt{Env}, \texttt{Sto}\rangle \to (\texttt{Env}, \texttt{Sto}, [V])}$$

$$\texttt{[if-true]:} \quad \frac{\mathcal{B}(C, \texttt{Env}, \texttt{Sto}) = \texttt{true} \qquad \langle P_T, \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}', \texttt{Sto}', \mathcal{O}\big)}{\langle \texttt{ifelse}(C, P_T, P_E), \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}, \texttt{Sto}', \mathcal{O}\big)}$$

$$\texttt{[if-false]:} \quad \frac{\mathcal{B}(C, \texttt{Env}, \texttt{Sto}) = \texttt{false} \qquad \langle P_E, \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}', \texttt{Sto}', \mathcal{O}\big)}{\langle \texttt{ifelse}(C, P_T, P_E), \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}, \texttt{Sto}', \mathcal{O}\big)}$$

$$\texttt{[function]:} \quad \frac{\langle \texttt{def}(F, A_{\text{formal}}, B), \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}', \texttt{Sto}', []\big)}{\langle \texttt{function}(F, A_{\text{formal}}, B), \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}', \texttt{Sto}', []\big)}$$

$$\texttt{[call]:} \quad \frac{\begin{array}{c}\mathcal{E}(N, \texttt{Env}, \texttt{Sto}) = (A_{\text{formal}}, Body, \texttt{Env}') \\ \big\langle \texttt{def}(A_{\text{formal}}, A_{\text{actual}}), \texttt{Env}', \texttt{Sto}\big\rangle \to \big(\texttt{Env}'', \texttt{Sto}'', []\big) \\ \big\langle Body, \texttt{Env}'', \texttt{Sto}''\big\rangle \to \big(\texttt{Env}''', \texttt{Sto}''', \mathcal{O}\big)\end{array}}{\langle \texttt{call}(N, A_{\text{actual}}), \texttt{Env}, \texttt{Sto}\rangle \to \big(\texttt{Env}, \texttt{Sto}''', \mathcal{O}\big)}$$

Figure 3.3. MiniLog Operational Semantics.

This semantics uses the auxiliary functions shown in Figure 3.4. We use $\mathcal{E}(E, S)$ to denote the evaluation of expression E under scope S. Arithmetic expressions in MiniLog allow addition (e.g. $add(x, y) = x + y$) and negation (e.g. $neg(x) = -x$) of integer constants and variables, and evaluate to an integer. Boolean expressions also operate on integers, following the common convention of considering 0 as the only false value, and all other values as true. Boolean expressions consist of equality comparisons (e.g. $eq(x, y) = 1$ if and only if $x = y, 0$ otherwise) and logical conjunction (e.g. $and(x, y) = x \wedge y$). Other common operations can be composed out of these two.

Given MiniLog's syntax, Definition 3.1 revisits the concept of *Context-Sensitive Optimization*. According to this definition, if a compiler implements a context-sensitive optimization O, then whenever context c becomes active during the execution of the program, the body of the active function is $O(c)$. Put it in another way, a context-sensitive optimization maps program contexts to clones of specialized functions. Ex-

$$Next(\texttt{Env}) = \begin{cases} 1 & \text{if } \texttt{Env} = [] \\ 1 + Next(\texttt{Env}') & \text{if } \texttt{Env} = [(N, L), \texttt{Env}'] \end{cases}$$

$$DefVar(N, V, \texttt{Env}, \texttt{Sto}) = \begin{array}{l} (\texttt{Env}', \texttt{Sto}') \\ \texttt{where} \\ \quad \texttt{Env}' = [(N, Next(\texttt{Env}))\,, \texttt{Env}] \\ \quad \texttt{Sto}' = [(Next(\texttt{Env}), V)\,, \texttt{Sto}] \end{array}$$

$$Location(N, \texttt{Env}) = \begin{cases} L & \text{if } \texttt{Env} = [(N, L)\,, \texttt{Env}'] \\ Location(N, \texttt{Env}') & \text{if } \texttt{Env} = [(N', L)\,, \texttt{Env}']\,, N \neq N' \end{cases}$$

$$SetVar(N, V, \texttt{Env}, \texttt{Sto}) = (\texttt{Env}, \texttt{Sto}')\ [(Location(N, \texttt{Env}), V)\,, \texttt{Sto}]$$

Figure 3.4. Functions used to define scope-related operations in MiniLog's semantics.

ample 3.2 illustrates these observations.

Definition 3.1 (Context-Sensitive Optimization in MiniLog)**.** A *context-sensitive optimization* $O : Context \mapsto P$ is a partial function that maps an optimizable context to the optimized body of the function that should run in that context. In this definition, we let `program`(P) be a syntactically valid MiniLog program.

Example 3.2. Figure 3.1 (b) shows the calling context tree of the program that appears in Figure 3.1 (a). We have two contexts which are amenable to be optimized by constant propagation: $C_0 = c_\emptyset \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$, and $C_1 = c_\emptyset \rightarrow c_1 \rightarrow c_2 \rightarrow c_4$. In the first context, C_0, we can replace the call of function a by a statement that prints the constant 0. In C_1, we can do the same, but for the constant 1. Thus, we have that $O(C_0) =$ function(a_0, x, print(0);), and that $O(C_1) =$ function(a_1, x, print(1);).

3.2 Optimization Trees

In this section, we present our main contribution: a method for generating minimal code that implements a given context-sensitive optimization (Definition 3.1). We use MiniLog for our presentation. We assume the instructions in a program P are numbered, in such a way that every instruction has a unique number, and we refer to the i-th instruction in P as P_i. Also, for this section, we assume that there is a single static path of call sites that reaches each optimized context. This class of contexts is formalized in Definition 3.2.

Definition 3.2 (Complete contexts)**.** A context C is a *complete context* of a MiniLog program P if its first `call` instruction not contained in the body of any function declaration[1].

Example 3.3. The context formed by the calls at lines $15, 11, 6$, reached during the execution of the program in Figure 3.1 is complete, as its first `call` instruction (at line 15) lays outside the scope of any other function.

We can widen this definition, allowing contexts to denote only a suffix of the required sequence of calls needed to activate a function. In Section 3.8, we discuss how we have adapted the presented solution to accommodate contexts which are not complete (*partial contexts*) in an efficient manner. However, for the sake of simplicity, our next developments assume only *complete contexts*.

Now we move on to define *Optimization Trees*. This data structure shall be necessary to guide the algorithm that implements our context-sensitive optimizations.

Definition 3.3 (Optimization Trees)**.** Let O be a context-sensitive optimization for a program P, and $\mathcal{D}(O)$ be the domain of O, i.e., the contexts for which some optimization applies. An optimization tree $T(O)$ is a graphic interpretation of $\mathcal{D}(O)$. Each vertex of this tree represents a distinct context. Edges correspond to `call` instructions. The root of the tree is the empty context $C_\emptyset$, where execution begins.

The Optimization Tree is a subgraph of the Calling Context Tree, and its nodes are all contexts that are either optimized by O or that lie on a path to an optimized context. Example 3.4 sheds some light on this definition.

Example 3.4. We have highlighted, in Figure 3.1 (b), the optimization tree that exists embedded in the calling context tree of the program in Figure 3.1 (a).

The nodes of an Optimization Tree are all contexts which are relevant to the application of the optimization. Because not only the optimized contexts (those in $\mathcal{D}(O)$), but also all their prefixes are in $T(O)$, a `call` instruction in the original program P may only cause the calling context to change inside $T(O)$ or to leave $T(O)$. However, once the current context is not in $T(O)$, no sequence of further call instructions can bring the program back to a context in $T(O)$ – `return` operations, on the other hand, can. Theorem 3.1 summarizes this result.

[1]This definition refers to MiniLog specifically. In languages where the entry-point is the `main` function, as in C or Java, a *complete context* would be a context where the first call instruction is in function *main*.

Theorem 3.1. *Let P be a MiniLog program. Let $C \to C'$ be a transition between two calling contexts that may happen when the execution of P reaches a `call` instruction P_i. For all possible C and C', exactly one of the following three cases is true:*

1. *Both C and C' are nodes of $T(O)$;*
2. *C is a node of $T(O)$, and C' is not;*
3. *Neither C nor C' are nodes of $T(O)$.*

Proof. We proceed by contradiction. Suppose that the execution of P may transition from a calling context C to some other context C' such that C is not a node of $T(O)$ but C' is. If such transition is possible, context C' must consist of exactly C with one appended call instruction: P_i. Thus, C must be a prefix of C'. By hypothesis, C' is a node of $T(O)$, and by the construction of $T(O)$ so must be C, which is a contradiction. Therefore, such case is impossible. $\square$

3.3 Cloning-based Code Generation

Given a context-sensitive optimization O and its corresponding Optimization Tree $T(O)$ to be applied to a program P, one classical method for generating a program P' that implements O is solely based on creating specialized function clones for each context in $T(O)$ (Hall [1991]). We start in top-level function calls (those not contained in the body of any function). Each such call that corresponds to an edge in $T(O)$ leaving the root either leads to an optimized context or is contained in a path that leads to one. If the call directly leads to an optimized context, then we declare the specialized function given by O for that context and change the call target so that the optimized procedure is called. Otherwise, we declare a clone of the target function and proceed recursively transforming the body of the newly created clone.

Algorithm 1 formalizes this method, which we call *Full Cloning*. We use this classical technique as a baseline in our experiments in Chapter 4. It iterates over all optimized contexts, i.e. those in $D(O)$. For each such context, it then goes over the calls that lead to that context. When a call reaches an optimized context, it replaces the target of the call by the optimized function. This happens in the end of optimization paths. When the next context in the path is not optimized, it checks whether a clone has already been created for the next context. If so, then the path has already been modified up to the current call. Otherwise, the algorithm creates a proper clone and modifies the function call leading to the next context to call the clone instead. Example 3.5 shows an example of the output of Full Cloning.

```
Data: Program $P$, Context-sensitive optimization $O$
Result: Optimized program $OptP$
$OptP \leftarrow P$;
/* Map from context to specialized clone (initially empty).   */
$ContextToClone \leftarrow$ new $Map$ Context $\mapsto$ Function;
/* Iterate over optimized contexts.                           */
forall $Context \in \mathcal{D}(O)$ do
    /* Iterate over edges in the context's call path.         */
    $CurrentContext \leftarrow \langle\rangle$;
    for $i \in 1...|Context|$ do
        $NextContext \leftarrow Concat(CurrentContext, Context[i])$;
        /* If NextContext is optimized, declare the optimized
        function and replace the call target.                 */
        if $NextContext \in D(O)$ then
            $Target \leftarrow O(NextContext)$;
            $TargetName \leftarrow$ ContextName$(NextContext)$;
            $OptP \leftarrow$
             InsertFunctionDeclaration$(OptP, TargetName, Target)$;
            $TransitionCall \leftarrow$ GetCall$(OptP, CurrentContext \rightarrow$
             $NextContext)$;
            $OptP \leftarrow$
             ReplaceCallTarget$(OptP, TransitionCall, TargetName)$;
        end
        else
            /* Otherwise, replace the call target by a clone.
            First, ensure a proper clone has been created.     */
            if $NextContext \notin ContextToClone$ then
                $Clone \leftarrow$ CloneFunction$(ActiveFunction(NextContext))$;
                $ContextToClone[NextContext] \leftarrow Clone$;
                $TargetName \leftarrow$ ContextName$(NextContext)$;
                $OptP \leftarrow$
                 InsertFunctionDeclaration$(OptP, TargetName, Clone)$;
            end
            $TargetName \leftarrow ContextName(NextContext)$;
            $TransitionCall \leftarrow$ GetCall$(OptP, CurrentContext \rightarrow$
             $NextContext)$;
            $OptP \leftarrow$
             ReplaceCallTarget$(OptP, TransitionCall, TargetName)$;
        end
        $CurrentContext \leftarrow NextContext$;
    end
end
```

Algorithm 1: The classical *Full Cloning* method for implementing a context-sensitive optimization.

Example 3.5. Figure 3.5 shows an example of the application of Full Cloning. In this case, the Optimization Tree in Figure 3.1(b) is applied to the program in Figure 3.1(a). Note that the original function `c` is never called in this program. In our actual implementation, LLVM gets rid of this procedure from the output binary when performing global dead code elimination. The original function `b`, on the other hand, is actually called from `c_0`, since the second call in `b` is not part of the optimization tree.

This example also shows the main issue with Full Cloning which our DFA-based code generation method tackles. In Figure 3.5, functions `b` and `b_c0` are not fundamentally different: they only call different versions of function `a`. Strictly speaking, `b` is not optimized in any way. However, Full Cloning needs both copies of `b` in order to differentiate between the calling paths that lead to optimized versions of `a` and those that do not. In longer paths, Full Cloning creates a copy of all functions in the path so that the end of path leads to an optimization.

3.4 DFA-based Code Generation

The cases in Theorem 3.1 are the basis of our method to track calling contexts at runtime. We keep track of the contexts that form the Optimization Tree of a given context-sensitive optimization O. For each optimization tree, we produce a state machine, and generate function clones. Notice that one optimization tree can represent the combination of several different compiler optimizations, such as constant propagation, type specialization, pointer disambiguation, etc. In other words, we shall produce one state machine per program – not one state machine per program optimization.

Code generation is based on a Deterministic Finite Automaton (DFA) derived from $T(O)$. In this section, we show how to transform the program assuming that we have the state machine. Section 3.5 shows how to build the DFA. Here, we present what we call the *outline implementation* of DFA-based context tracking. In Section 3.7, we present the *inline implementation*, borrowing the same ideas. The difference is that the outline implementation inserts calls to a `transition` function in the program, that is assumed to implement DFA transitions (in Section 3.5 we show how to implement `transition`). The inline implementation does not rely on any additional function calls, adding code to handle the DFA inline in relevant functions of the program.

Algorithm 2 transforms an input program P to apply a context-sensitive optimization O, optimizing program P in-place. It has the following phases:

1. Declares the `transition` function, which queries and updates the implicit state machine. Section 3.5 shows how to build the DFA and implement `transition`.

```
function(a, x,
  print(x);
);

function(a_b0_c0, x,
  print(0);
);

function(a_b1_c0, x,
  print(1);
);

function(b, x,
  call(a, 2*x);
  call(a, 2*x + 1);
);

function(b_c0, x,
  call(a_b0_c0, 2*x);
  call(a_b1_c0, 2*x + 1);
);

function(c, x,
  call(b, 2*x);
  call(b, input);
);

function(c_0, x,
  call(b_c0, 2*x);
  call(b, input);
);

call(c_0, 0);
```

Figure 3.5. Program from Figure 3.1 optimized with *Full Cloning.*

For now, the following assumptions on `transition` are made:

- `transition` takes one parameter: the input to the current state of the DFA. It can be either an identifier of a call instruction, or -1 (for signaling the return operation).
- Its output is a reference to the function that should be called in the state after the transition, and it is stored in the global variable sm_function. Depending on the current state (calling context), the returned function might or might not be optimized.

2. Declares the optimized functions given in O, naming them after the context they should be called in.
3. Declares one copy of each function that appears in at least one context in the Optimization Tree. This avoids invoking the state machine within the body of functions that do not belong into the optimization tree. At most one copy of each function is made, even if it appears several times in $T(O)$. The names of these copies are the names of the original functions added to the "ctx_tr_" prefix.
4. Modifies function calls that correspond to edges in $T(O)$ to update the state machine before and after calling the target function. To be modified, a function call must correspond to an edge in the Optimization Tree. Moreover, it must be either a top-level instruction (i.e. outside of all functions), or contained in one of the context-tracking functions created in phase 3. Modified calls are replaced by a sequence of 3 function calls: one to move the state machine to the next state and retrieve the function to be called, one to call the target function, and one to return the state machine to the previous state.

The main advantage of using Algorithm 2 to apply an optimization is that the number of copies of functions made is exactly the number of *distinct* functions that appear in the Optimization Tree. Should Full Cloning or inlining be used, a function's body would be copied once for each of its occurrences in the Optimization Tree. In this way, we minimize code duplication, using a data structure to track the calling context, instead of context-specific copies of code (on which both cloning and inlining are based).

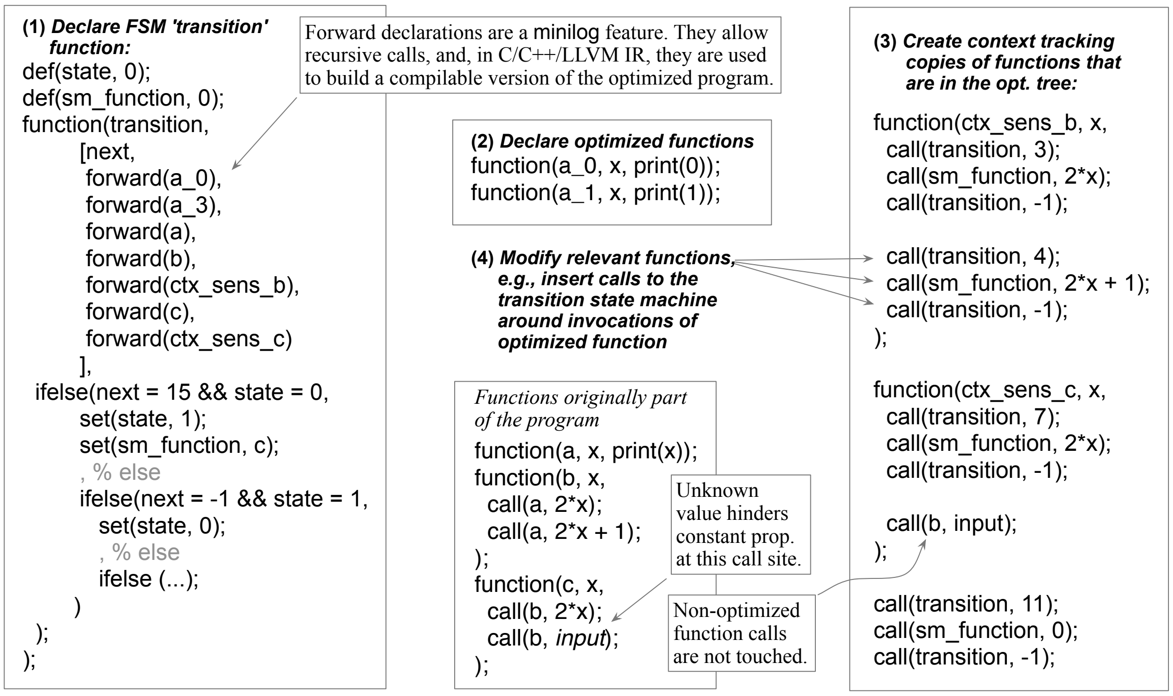
Example 3.6. Figure 3.6 shows the code that Algorithm 2 produces for the program in Figure 3.1 (a), using the optimization tree in Figure 3.6 (b). The state machine produced in this case appears in Figure 3.6 (c). We have produced two clones of function `a`, called `a_0` and `a_1`, one for each optimized context.

```
Data: Program $P$, Context-sensitive optimization $O$
Result: Optimized program $OptP$
/* (1) Declare DFA ‘transition’ function.                    */
$OptP \leftarrow DeclareTransitionFunction(P, O)$ ;
/* (2) Declare optimized functions.                          */
forall $Context \in \mathcal{D}(O)$ do
    $OptP \leftarrow InsertFunctionDeclaration(OptP, ContextName(C), O(C))$;
end
/* (3 - Optional) Create 1 ‘context-tracking’ clone for every
path function in $T(O)$.                */ $OptTree \leftarrow T(O)$ ;
forall $Node \in OptTree$ do
    $CtxTrackingFuncName =$ “$ctx\_tr\_$" $+ Node.Function.Name$;
    if not $IsDeclared(OptP, CtxSensitiveName)$ then
        $OptP \leftarrow$
          $InsertFunctionDeclaration(P, CtxTrackingFuncName, Node.Function.Body)$;

    end
end
/* (4) Modify relevant function calls to track calling context.
*/
forall instruction $P_i \in OptP$ do
    if $IsOptTreeEdge(OptTree, i)$ and
    $(IsTopLevelInstruction(P, i)$ or $IsInCtxTrackingFunction(P, i))$ then
        $OptP \leftarrow Replace(OptP, P_i,$[
        call$(transition, i)$,
        call$(sm\_function, CallArg(P_i))$,
        call$(transition, -1)$
        ]);
    end
end
```

Algorithm 2: Code generation procedure that populates the program with calls to the DFA that tracks contexts.

Avoiding the overhead on non-optimized functions Our state machine lets us clone only leaf functions, i.e., code that can actually be specialized. However, in this case, it would be necessary to surround with state transitions every function invocation. This would impose an unnecessary overhead upon the parts of the program that were not optimized. To avoid this overhead, we produce one, and only one, clone of every path function that is in the optimized tree. This clone, which bears the prefix `ctx_sens_`, contains the machinery to switch states in our DFA. Such functions are produced by step (3) of Algorithm 2. We refer to these functions as *auxiliary path clones*.



```
(1) Declare FSM 'transition'
    function:
def(state, 0);
def(sm_function, 0);
function(transition,
        [next,
         forward(a_0),
         forward(a_3),
         forward(a),
         forward(b),
         forward(ctx_sens_b),
         forward(c),
         forward(ctx_sens_c)
        ],
  ifelse(next = 15 && state = 0,
        set(state, 1);
        set(sm_function, c);
        , % else
        ifelse(next = -1 && state = 1,
           set(state, 0);
           , % else
           ifelse (...);
        )
  );
);
```

Forward declarations are a minilog feature. They allow recursive calls, and, in C/C++/LLVM IR, they are used to build a compilable version of the optimized program.

```
(2) Declare optimized functions
function(a_0, x, print(0));
function(a_1, x, print(1));
```

(4) Modify relevant functions, e.g., insert calls to the transition state machine around invocations of optimized function

```
Functions originally part
of the program
function(a, x, print(x));
function(b, x,
  call(a, 2*x);
  call(a, 2*x + 1);
);
function(c, x,
  call(b, 2*x);
  call(b, input);
);
```

Unknown value hinders constant prop. at this call site.

Non-optimized function calls are not touched.

```
(3) Create context tracking
    copies of functions that
    are in the opt. tree:

function(ctx_sens_b, x,
  call(transition, 3);
  call(sm_function, 2*x);
  call(transition, -1);

  call(transition, 4);
  call(sm_function, 2*x + 1);
  call(transition, -1);
);

function(ctx_sens_c, x,
  call(transition, 7);
  call(sm_function, 2*x);
  call(transition, -1);

  call(b, input);
);

call(transition, 11);
call(sm_function, 0);
call(transition, -1);
```

Figure 3.6. Code produced by Algorithm 2, to implement constant propagation on the program seen in Figure 3.1 (a).

Example 3.7. Figure 3.6 contains two auxiliary path clones: ctx_sens_b and ctx_sens_c. We also kept the original versions of functions b and c. The original, untouched, functions are used in contexts $C_1 \to C_5 \to C_6$ and $C_1 \to C_5 \to C_7$ (see Figure 3.1), which could not be optimized by constant propagation. Notice that, independent on the optimization tree, we would not produce more versions of ctx_sens_b and ctx_sens_c.

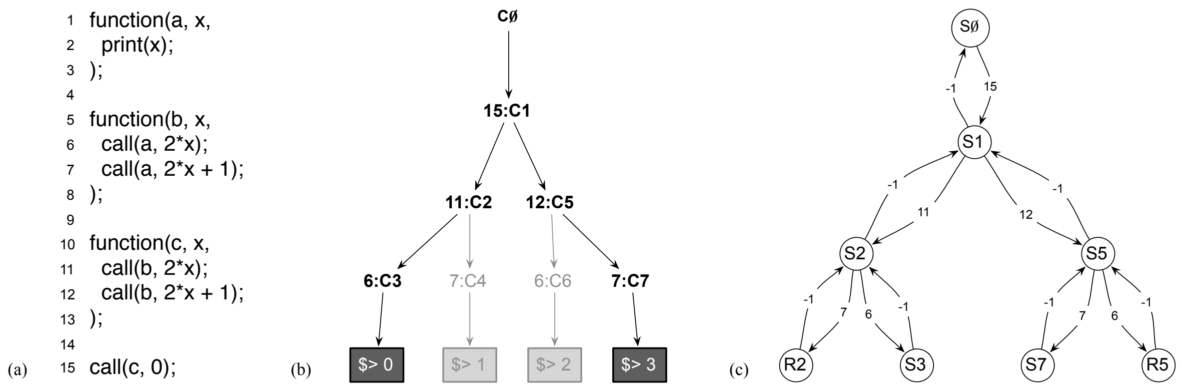
3.5 Construction of the Finite State Machine

The cases in Theorem 3.1 are our basis for building an DFA derived from $T(O)$. Algorithm 3 builds the DFA needed by Algorithm 2. In phase 1, every node of $T(O)$ becomes a state in the DFA. Phase 2 covers transitions that fall into Case 1 of Theorem 3.1. Call instructions that correspond to edges in $T(O)$ become transitions in the DFA, and their identifier in the program matches the corresponding transition's label in the DFA. Moreover, the DFA has transitions that correspond to returning from function calls, which always have the same sentinel label, -1, regardless of the context. Case 3 of Theorem 3.1 is handled implicitly. Algorithms 2 and 3 are completely oblivious to transitions between contexts not in $T(O)$; hence, they are processed as in the original program.

Calls that force the program flow to leave the Optimization Tree (Case 2 in Theorem 3.1) may be further divided into two classes. Some call instructions never cause transitions inside $T(O)$, i.e. no edge in $T(O)$ corresponds to them. We shall name these calls *context-insensitive*. They are context-insensitive with regards to the optimization described by $T(O)$. Such calls are not modified by Algorithm 2; thus, the context-insensitive version of the callee is invoked. In other words, context-insensitive functions cannot provoke transitions in the DFA. Therefore, when a context-insensitive function returns, the DFA will be in the state reached by the last caller in $T(O)$. The second category of functions that might cause the program to leave $T(O)$ are called *context-semisensitive*. They may cause the program to leave $T(O)$ in some contexts, but transition inside $T(O)$ in others. Such calls require special care. Example 3.8 illustrates this concept.

Example 3.8. Figure 3.7 (a) shows the program earlier seen in Figure 1.1, written in MiniLog. Contrary to the MiniLog program in Figure 3.1, every invocation of function a, in this new version, could be fully optimized by constant propagation. However, for the sake of the example, we assume that only contexts $C_\emptyset \rightarrow C_2 \rightarrow C_3$ and $C_\emptyset \rightarrow C_5 \rightarrow C_7$ are optimized. The optimization tree appears in Figure 3.7 (b). This assumption gives us the opportunity to show how we navigate in and out of the optimization tree during the execution of the program. In Figure 3.7 (a), the call in line 6 is an example of a context-semisensitive invocation. When the program is in context C_2, the call from line 6 leads it to context C_3, which is optimized. However, in context C_5, the same call leads the program flow to context C_6, which is not part of the optimization tree. In the state machine, seen in Figure 3.7 (c), this unoptimized context is represented as state R5. Once this unoptimized invocation of b returns, the transition key -1 moves the active state back to state S5.

Going back to Example 3.8, to handle the transition between C_5 to C_6, we create one special state R5 which has a single transition, returning to S5. Such state is called a *return-only state*. When a transition to a *return-only* state happens, the context-insensitive version of the target function is called. While that function runs, the DFA sits in the *return-only* state. When the call returns, because the caller is a *context-sensitive* function, it will feed the current state the -1 sentinel input. This transition key causes the DFA to return to the last state visited while the program was within the optimization tree. This scheme, implemented in Phase 3 of Algorithm 3, completes our coverage of all possible transition types in Theorem 3.1.



```
1  function(a, x,
2    print(x);
3  );
4
5  function(b, x,
6    call(a, 2*x);
7    call(a, 2*x + 1);
8  );
9
10 function(c, x,
11   call(b, 2*x);
12   call(b, 2*x + 1);
13 );
14
15 call(c, 0);
```

Figure 3.7. (a) Example of program that can be fully optimized with constant propagation. (b) Optimization Tree that we obtain, assuming that only the two highlighted contexts are optimized. (c) The DFA derived from the Optimization Tree. In the DFA, a *return-only* state corresponding to state C_i is labeled R_i. These states are created by step (3) of Algorithm 3.

The `transition` function used by Algorithm 2 is implemented in MiniLog as a sequence of `ifelse` statements, that test all $(CurrentState, Input)$ pairs. In our LLVM implementation, `transition` is implemented using switch statements, which run in $O(1)$ Korobeynikov [2007]. First, a switch statement identifies the current state. Then, another nested switch statement acts upon the transition key. All state and transition identifiers are mapped beforehand to contiguous integer ranges to speed-up this implementation. The function returned by `transition` depends on the type of the target state. If the state of the DFA after the transition is a node of $T(O)$, then either the optimized function (given by O, in Definition 3.1) or the context-sensitive version of the callee is returned. If a *return-only* state is reached, then `transition` returns the original (context-insensitive) version of the callee.

3.6 Properties of DFA-based Code Generation

Given a context-sensitive optimization O to be applied onto a program P, we use Algorithm 3 to generate the DFA that supports the application of O, and use Algorithm 2 to transform the original source code. This transformation gives us an optimized program that implements O. Our algorithm creates one clone for each leaf function, and at most one clone for any path function. Path functions are cloned only once, regardless of how many times they appear in the optimization tree. We emphasize that the creation of this clone is *optional* – we do it to avoid imposing on non-optimized functions the burden of changing the DFA. Property 3.2 puts a bound on the number of clones that we produce for a given optimization tree.

Property 3.2. *Algorithm 2 creates one clone per leaf function in $T(O)$. Optionally, it creates one auxiliary path function to implement all the occurrences of path functions that have the same name.*

Proof: Algorithm 2 creates one clone for every element in $\mathcal{D}(O)$, the domain of the context-sensitive optimization. This event happens in step (2) of the algorithm. Because each function in $\mathcal{D}(O)$ can be specialized in a different way, all these clones are necessary. *Optionally*, Algorithm 2 creates at most one auxiliary path function for each path function in $T(O)$, i.e., a function that leads to specialized code, but that does not belong into $\mathcal{D}(O)$. Thus, each non-specialized function might have up to two clones in the final program, independent on how many times it is called in the code. $\square$

The generated `transition` function will contain a representation of the entire Optimization Tree; hence, the DFA's size can be exponential. However, the DFA is equivalent to a perfect hash-table. Thus, contrary to unrestricted cloning, or unrestricted inlining, it is a data structure (which represents the state machine), not code that grows. As Chapter 4 shows, tracking contexts using this method lets us have a context-sensitive optimization that scales to programs having up to 10^{16} contexts. Property 3.3 summarizes this fact.

Property 3.3. *Algorithm 3 produces a DFA with at most* 2 *states per node in* $T(O)$

Proof: In step (2), Algorithm 3 creates one state for every node in $T(O)$. In step (3), Algorithm 3 might create one *return-only* state for each node in $T(O)$. $\square$

We prove that the combination of Algorithms 2 and 3 produces correct programs via *bisimulation*. To this end, we let $P' = CS(P, O)$ be the result of applying this code transformation onto a MiniLog program P, given a context-sensitive optimization O. Theorem 3.4 shows that the transformation is correct. Supporting Lemmas 3.1, 3.2 and 3.3 prepare the ground for that theorem. The first of these lemmas shows that there exists an isomorphism between the calling context trees of the original and transformed programs.

Lemma 3.1 (CCT Isomorphism)**.** Let P be a MiniLog program, O be a Context-Sensitive Optimization, and $P' = CS(P, O)$. Function f is invoked at context C_i in program P, if, and only if, there exists context C'_i, such that, function $O(f, C_i)$ is invoked at context C_i in program P'.

Proof: By induction on the *call string* corresponding to C_i. The call string is a sequence of function calls that represent the state of the invocation stack at any given point of the program's execution. Consider call string of length K. If $K = 0$, the lemma holds trivially, as we have a Calling Context Tree with only one node. If $K > 0$, then we let C_k be a context whose call string has K elements, and C_{k-1} the context from which C_k has been reached. We let f_p be the function active at C_{k-1}. By induction, the lemma is true for C_{k-1}. There are four cases to consider, when analyzing C_k:

1. $C_{k-1} \notin T(O)$ and $C_k \notin T(O)$: in this case, $O(f_p, C_{k-1}) = f_p$, $O(f, C_k) = f$, and P and P' experiment the same transition.
2. $C_{k-1} \notin T(O)$ and $C_k \in T(O)$: by Theorem 3.1, this case is impossible.
3. $C_{k-1} \in T(O)$ and $C_k \notin T(O)$: transition moves the DFA from a state s to a return-only state s_r, returning the original, context-insensitive version of f. The return-only state s_r is unique. So, the same call will be performed in P and P'. The call transition(-1) will bring the DFA back to s.
4. $C_{k-1} \in T(O)$ and $C_k \in T(O)$: one transition in the DFA happens. Function transition is invoked in P' in a state $s_{C_{k-1}}$ that corresponds uniquely to C_{k-1}. Because each context in $T(O)$ directly maps to one state in the DFA, necessity and sufficiency hold. Upon function return, P returns to C_{k-1}, and the DFA returns to $s_{C_{k-1}}$, due to the semantics of transition(-1). □

For stating and proving Lemmas 3.2 and 3.3, we use the notion of a *projection*, which Definition 3.4 describes. Projections are necessary for correctness proofs, because the state of the original program is a subset of the state of the optimized program.

Definition 3.4 (Projection)**.** Let F_1 and F_2 be two functions. We say that $F_1 <: F_2$ if, and only if, for any x in the domain of F_1, $F_1(x) = F_2(x)$.

Lemma 3.2 (Step in the Original Program)**.** Let $\texttt{program}(P)$ be a MiniLog program, and $CS(P, O) = P'$. If $\langle P, \texttt{Env}, \texttt{Sto}\rangle \to (\texttt{Env}', \texttt{Sto}')$, then there exists an environment $\texttt{Env}''$, and a store $\texttt{Sto}''$, such that $\langle P', \texttt{Env}, \texttt{Sto}\rangle \to (\texttt{Env}'', \texttt{Sto}'')$, where $\texttt{Env}' <: \texttt{Env}''$, and $\texttt{Sto}' <: \texttt{Sto}''$.

Proof (sketch): Every instruction in P exists also in P', with the exact same arguments, except for function calls. Function calls in P and P' might invoke different targets, if the one in P' belongs into $T(O)$: in P we have $\texttt{call}(f, x)$, and in the latter $\texttt{call}(\text{sm\_function}, x)$. From Lemma 3.1, we know that these calls lead to corresponding contexts, e.g., C and C', and we know that $\text{sm\_function} = O(f, C)$. The proof follows from two assumptions: (i) the context-sensitive optimization yields a semantically equivalent version of F, which is assigned to sm_function; (ii) the extra instrumentation produced by $C(P, O)$ does not write in variables originally present in P. $\square$

Lemma 3.3 (Step in the Transformed Program)**.** Let program(P) be a MiniLog program, and $CS(P, O) = P'$. If $\langle P', \texttt{Env}, \texttt{Sto}\rangle \rightarrow (\texttt{Env}', \texttt{Sto}')$, then there exists an environment $\texttt{Env}''$, and a store $\texttt{Sto}''$, such that $\langle P, \texttt{Env}, \texttt{Sto}\rangle \rightarrow (\texttt{Env}'', \texttt{Sto}'')$, where $\texttt{Env}'' <: \texttt{Env}'$, and $\texttt{Sto}'' <: \texttt{Sto}'$.

Proof (sketch): This proof is similar to the one seen in Lemma 3.2; however, it must deal with a caveat: P' contains instructions that find no correspondent in P. These instructions consist on invocations of the transition function, and the operations present in the body of transition. None of these operations interfere with variables defined in either $\texttt{Env}''$ or $\texttt{Sto}''$. Therefore, they preserve the invariant $\texttt{Env}'' <: \texttt{Env}'$, and $\texttt{Sto}'' <: \texttt{Sto}'$. If P' steps via one of those instructions, then it suffices to apply the empty step onto P. $\square$

Theorem 3.4 (Correctness)**.** *The transformation* CS *preserves the semantics of MiniLog programs.*

Proof: The theorem follows as a corollary of Lemma 3.2 and Lemma 3.3. $\square$

3.7 Inline DFA Implementation

In this section, we show an alternative form of implementing an Optimization Tree using a dynamically updated DFA, which we call the *inline implementation* of DFA-based code generation.

Algorithm 2 assumes the existence of a function called `transition` that is responsible for changing the states of the DFA upon function calls and returning a reference to the implementation that should be invoked in the current calling context. This incurs the cost of two procedure calls for each transition the DFA performs. Moreover, based on the implementation of `transition` described in Section 3.5, each invocation of this function runs over two switch statements: one for checking which state the DFA is currently at (a global variable), and one for checking which transition should be made based on the identifier of the current call site (the single parameter passed to `transition`).

We can, however, bring the cost per transition down to a single switch statement instead of two, and avoid the function calls altogether. Suppose we inlined the huge `transition` function everywhere it is called in the optimized program with a parameter different than -1, i.e. once in all call sites that may trigger a DFA transition. Now, consider one such call site CS, in the body of some function f. The second (nested) switch statement in the body of `transition` checks which call site we are at. However, since we inlined the implementation of `transition` in the call site CS, we do not need such check, since this in this call site the switch statement will always select CS. Thus, we can simply remove this switch and replace it by the body of its "CS" case. We now have a single switch statement which tests for the DFA state. It considers the possibility of the DFA being in each of its states. Here comes the key observation that makes this approach practical: given that the call site CS is located in function f, we know that not all states are possible. Rather, only the DFA states in which function f is active are possible, which is usually a very small subset of all states. Thus, we can remove all other states from the switch statement. We are still left with the call to `transition(-1)`. We can avoid this one by simply noting that it will always bring the DFA to the state it was at when the execution reached function f. Therefore, in the beginning of f we can save the current DFA state in a local variable st. Then, all calls to `transition(-1)` can be replaced by a statement assigning the global state back to the state saved in st. This inline implementation usually needs more space than the outline one, since the set of states in which a given function is active is repeated in all call sites in that function that trigger DFA transitions. In contrast, in the outline implementation from Section3.4, every such call site brings a constant additional overhead (of calling `transition`).

Example 3.9. Figure 3.8 shows function `b` from Figure 3.7 optimized with the inline DFA implementation described in this section. The applied Optimization Tree is also the one from Figure 3.7. Given the contexts shown in Figure 3.1, we know that if the program is currently at `b`, there are only two possible states: $S2$ and $S5$. Thus, for each of the two transitions, we only need to check which state we are at in order to know the next state and the function to be called.

3.8 From MiniLog to LLVM's Intermediate Representation

We have implemented the ideas discussed in this section in LLVM. This implementation is very similar to the one we have described in Algorithms 2 and 3; however, to make it practical, we have adopted two extensions, which we describe below. The first extension

```
function(b, x,
  def(st, __global_dfa_state);
  # Originally call(a, 2*x);
  ifelse(st == 2,
    [
      set(__global_dfa_state, 3);
      call(a_0, 2*x);
      set(__global_dfa_state, st);
    ],
    [
      call(a, 2*x);
    ]);

  # Originally call(a, 2*x + 1);
  ifelse(st == 5,
    [
      set(__global_dfa_state, 7);
      call(a_3, 2*x + 1);
      set(__global_dfa_state, st);
    ],
    [
      call(a, 2*x + 1);
    ]);

);
```

Figure 3.8. Function `b` from Figure 3.7 optimized with inline DFA.

allows us to support context-sensitive optimizations described by *partial contexts*. The second extension allows us to support exception handling in functions that use this feature.

Supporting partial contexts. Algorithms 2 and 3 assume that the root of an optimization tree is always the entry point of the program. However, an analysis might discover optimization opportunities starting in any function. For example, if some function f calls some function g with constant parameters, and these same values are forwarded from g to h, then h is optimizable every time it is reached from f. The path from the *main* function to f is not relevant in this case. Although we can represent this optimization in a single Optimization Tree by exhaustively listing all contexts in which f is called, that is undesirable, since f might be active in an exponential number of distinct contexts. In order to support describing such optimization efficiently, we can easily extend our method to handle *Optimization Forests* instead of *Optimization Trees*. An *Optimization Forest* associates an *Optimization Tree* with a collection of *root functions*. Whenever control flow reaches a *root function*, context-tracking begins relative to that function. Instead of having a single global DFA state, we shall have a stack of states. The state on the top corresponds to the currently active *Optimization Tree* and context-tracking DFA. *Root functions* are modified to push the initial state of their corresponding DFA to the stack upon invocation. When the function returns, the state is popped, returning to whatever Optimization Tree was active before (if any). Function *transition* will always operate on the top of the stack. In this way, context-tracking overhead only takes place when a *root function* is in the call stack. Otherwise, parts of the original program will be executed.

Supporting exception handling. To explain how we track calling contexts using a DFA, we have used a call to function transition with a sentinel parameter -1 to indicate function return. However, the same approach would not work in a language that supports exceptions. To handle exceptions with the *outline implementation* of the DFA, we have adopted a slightly different approach. Like the inline implementation (Section 3.7), we save the current state of the DFA in a local variable at the beginning of every *context-sensitive* function (i.e. those that update the DFA). Thus, the call to transition(-1) is avoided by simply copying back the locally saved state to the global DFA state. The same can be done to correct the DFA state whenever an exception is being handled (e.g. in a `catch` block in C++ or Java). Our actual LLVM implementation processes returns using this method, since it has also proven profitable in terms of minimizing the overhead involved in updating the state machine. This modification can work along with *Optimization Forests* by inserting a `catch-all` block in the end of every *root function*, which ensures the state is popped even if an exception is thrown.

```
Data: Context-sensitive optimization $O$
Result: Automaton $DFA$
$DFA \leftarrow$ new $Finite\ State\ Machine$;
/* (1) Create states for nodes in $T(O)$.                          */
$OptTree \leftarrow T(O)$ ;
forall $Node \in OptTree.Nodes$ do
    $DFA.addState(Node.Label)$;
end
/* Map from Function to set of edges that cause at least one
transition inside $T(O)$,     */ /* in any context the function is
active in                                                          */
$TransitionSet =$ empty map : $Function \mapsto Set < Edge,\ label >$;
/* (2) Create transitions for calls/returns inside $T(O)$ (Case 1
of Thm. 3.1).                                                      */
forall $Node \in OptTree.Nodes$ do
    forall $Edge \in Node.EdgesToChildren$ do
        $DFA.addTransition(Node.Label, Edge.Destination.Label, Edge.CallInstLabel)$;

        $DFA.addTransition(Edge.Destination.Label, Node.Label, -1)$;
        $TransitionSet[Node.ActiveFun] \leftarrow$
          $TransitionSet[Node.ActiveFun] \cup \{Edge.CallInstLabel\}$;
    end
end
/* (3) Create return-only states & transitions that leave $T(O)$
(Case 2 of Thm. 3.1).                                              */
forall $Node \in OptTree.Node$ do
    /* List transitions that Node.ActiveFun might have in other
    nodes,                                                         */
    /* but not at the current node.  If there are any, we need a
    return-only state.                                             */
    $MissingTransitions \leftarrow TransitionSet[Node.ActiveFun] \setminus$
      $\{E.CallInstLabel : E \in Node.EdgesToChildren\}$;
    if $MissingTransitions \neq \varnothing$ then
        $ReturnOnlyStateLabel \leftarrow$ concatenate(“$R$”, $Node.Label$);
        $DFA.addState(ReturnOnlyStateLabel)$;
        $DFA.addTransition(ReturnOnlyStateLabel, Node.Label, -1)$;
        forall $MissingLabel \in MissingTransitions$ do
            $DFA.addTransition(Node.Label, ReturnOnlyStateLabel, MissingLabel)$;

        end
    end
end
```

Algorithm 3: Algorithm for building a Finite State Machine which tracks contexts that are relevant to a given context-sensitive optimization.

Chapter 4

Experimental Evaluation

In this chapter, we evaluate our context-sensitive code generation method when applied in a practical scenario. We have implemented both the inline and outline versions of method in LLVM 3.9.1. To evaluate our ideas experimentally, we need a concrete optimization, since our contributions are optimization-agnostic. To that end, we have implemented a fully context-sensitive version of constant propagation. The algorithm propagates constants inside functions and, whenever a parameter of a call instruction is discovered to be constant, propagation continues in the target function in a context-specific scenario. This information – discovered statically – is arranged in an Optimization Tree, which is then passed to our code generator. We do not optimize functions when the only change that constant propagation enables is to replace call arguments by constants. Rather, propagation continues in child contexts without the caller being modified.

We compare our methods against a clone-based context-sensitive code generator. In this baseline, each node of the Optimization Tree is implemented by a function clone. Every function in the optimization tree is replicated once per occurrence in the tree. Unoptimized functions are not touched; hence, this approach is similar to previous art (Li et al. [2013]). We compare both methods in terms of generated code size and running time, aiming to answer the following research questions:

1. Are the DFA-based code generation methods competitive in terms of final code size, when compared to traditional function cloning?

2. Is the run-time overhead introduced by the DFA small enough for it to be competitive with cloning, which has no run-time overhead?

Benchmark	**Functions**	**Instructions**	**Contexts**	**Opt. Contexts**	**Opt. Tree Nodes**	**DFA States**
mcf	19	2,256	106	14	23	46
gobmk	2,366	124,098	$> 10^{16}$	3,752	14,998	29,996
hmmer	196	33,185	28,880	2,396	3,904	7,808
libquantum	56	3,862	236,833	359	740	1,480
gcc	3,784	610,935	$> 10^{16}$	5,701	14,543	29,086
h264ref	405	145,757	4,441	642	1,185	2,370
astar	60	7,474	1,339	139	220	440
bzip2	43	21,904	2,905	245	316	632
perlbench	1,392	245,265	$> 10^{16}$	1,668	15,000	30,000
sjeng	100	21,982	1,388,357	955	14,437	28,874
omnetpp	1,552	78,176	6	0	0	0
lbm	15	2,240	102	8	10	20
povray	1,306	194,386	39	0	0	0
namd	89	62,722	139	0	0	0
milc	157	16,066	11,400	1,143	1,707	3,414
sphinx3	194	29,482	24,630	2,303	14,897	29,794
soplex	690	76,157	99	0	0	0

Table 4.1. Benchmarks in SPEC CPU2006. **Opt. Contexts** shows the number of contexts that could be optimized, out of 15,000 samples. This number is also the quantity of clones created by our approach. **Opt. Tree Nodes** gives us the number of nodes in the optimization tree. This number is the quantity of clones created by typical clone-based specialization.

4.1 Benchmarks

We tested our implementation on 191 benchmarks integrated in the LLVM Test Suite, which includes several well-known benchmarks such as MiBench, Stanford, Shootout and Polybench. We have augmented this suite with SPEC CPU2006 Henning [2006]; thus, obtaining 18 real-world benchmarks such as gcc, bzip2 and perl[1]. All tests were ran on an Intel Xeon CPU E5-2620 0 @ 2.00GHz CPU, with 16 GB of DDR3 RAM. Table 4.1 shows statistics about the SPEC CPU2006 benchmarks. Some benchmarks have considerably large code (as gcc, with 2.673 functions). We have also counted the number of static contexts in these programs using a simple dynamic programming algorithm on the call graph. This count ignores recursive functions (otherwise, the number of statically known contexts is theoretically infinite). Some programs have a very large number of contexts (over 10^{16} in a few cases).

Context Pruning. To handle such large search spaces, our context-sensitive constant propagation limits its search to a fixed number of contexts. In our experiments, we limited the number of analyzed contexts to 15,000. Table 4.1 also shows how many

[1]SPEC CPU2006 has 13 other benchmarks not integrated into the CMake-based setup of the LLVM Test Suite, which we use. The use of the old Makefile-based setup, which comprises those benchmarks, is discouraged by current documentation.

contexts were effectively optimized in each SPEC benchmark, and the number of nodes in the Optimization Tree that is later passed to code generation methods. The algorithm starts at the main function, propagating constants as deep as possible in the Context Call Tree. Once information in a context does not allow any further propagation, it returns and explores other nodes of the CCT. We could combine the algorithm with heuristics to prioritize contexts which are more likely to be profitable to optimize, e.g. by guiding the search using static profiling information. However, since the focus of this dissertation is not on the optimization itself, we did not do so. The algorithm naïvely analyzes the first contexts it reaches by following calls in the order they appear in the program. In total, this process takes about 4 hours to go over the 209 available programs. This time is mostly spent in the context-sensitive implementation of constant propagation. Code generation, i.e., construction of clones and the state machine, takes negligible time. Nevertheless, this simple optimization can yield speedups on large programs in SPEC CPU2006 when comparing to LLVM in the -O3 optimization level, as our experiments show.

Indirect Calls. Neither the counting algorithm nor the optimization tries to discover the targets of indirect function calls. Therefore, our implementation does not optimize dynamically invoked functions. In SPEC CPU2006, this especially affects `xlancbmk` and `omnetpp`. Because dynamic dispatch is used very early in the execution of these benchmarks, their CCT could not be accurately constructed at compile-time. Thus, the numbers in Table 4.1 reflect only the subtree of the CCT that can be processed ignoring indirect calls. We emphasize that this shortcoming in our implementation is not a limitation of our technique. Indirect calls affect every context-sensitive optimization, and there are techniques to deal with them Dean et al. [1995]; Milanova et al. [2004]; Shivers [1988].

4.2 Generated code size

Figure 4.1 compares the size of binaries produced by the outline implementation of the DFA-based code generation and by full cloning. On the horizontal axis, we have the 128 benchmarks in which actual optimizations took place (i.e. context-sensitive constant propagation has changed the code at all). The vertical axis shows the difference between the binary size generated between cloning and the outline DFA-based method (the former minus the latter). Thus, when the y-value of a benchmark is above zero, the DFA-based method generated smaller code for the benchmark (difference was positive). When it is negative, then cloning generated a smaller binary. The vertical dashed line marks the first benchmark in which cloning was worse. Note that it is far to the right: only in 31 of all 128 benchmarks has the DFA-based method generated a smaller binary. However, as it is evident from the graph, the gains far outweigh the losses: while in smaller cases cloning is slightly more space efficient, in larger cases it explodes. The larger the benchmark, the more likely it is that the DFA-based method will save space. Figure 4.2 shows similar results comparing inline implementation of the DFA against cloning. The inline DFA is less compact: the binaries generated with this method are larger than those generated with cloning on 102 benchmarks. Nevertheless, the largest differences all favor the DFA, like in the outline implementation.

When summing over all benchmarks, the binaries produced by full cloning are 2.76 times larger than the binaries that we generate with our outline state machines, and 3.61 times larger than the original binaries. If we only consider the bytes *added* to the original binary (i.e. optimized binary size minus binary size generated by LLVM -O3), the difference is even larger: while our method adds 33.37MB in total to all executables, cloning adds 281.94MB in order to implement the same optimization, i.e. it adds 8.5x more bytes. The largest relative difference between the two approaches was observed in Fhourstones-3.1: full cloning produced an executable more than 100x larger than our technique. However, sometimes our DFAs yield larger binaries. The largest discrepancy was observed in FreeBench/mason, where we have generated a binary 11x larger than full cloning. This situation is possible if the target program has a flat call graph, with few path functions. Under these circumstances, DFA and full cloning will produce roughly the same number of versions; however, we must account for the additional space taken by the state machine. Nevertheless, this experiment allows us to give our first research question a positive answer: results show the DFA-based method is able to implement context-sensitive optimizations more efficiently than traditional function cloning.

4.3 Performance of Generated Code

Figure 4.3 compares the running times of the executables produced by the outline DFA-based approach and full cloning. Numbers are the average of five executions. As in Figure 4.1 (a), all the benchmarks have been compiled with LLVM -O3. The figure makes it visually apparent that both the approaches yield programs that have similar run-times. In total, the binaries produced by LLVM -O3 took 1510.5s to run, versus 1515.48s from cloning, 1517.48s from the inline DFA and 1521.26s from the outline DFA-based method. On average, binaries produced with full cloning are only 0.4%

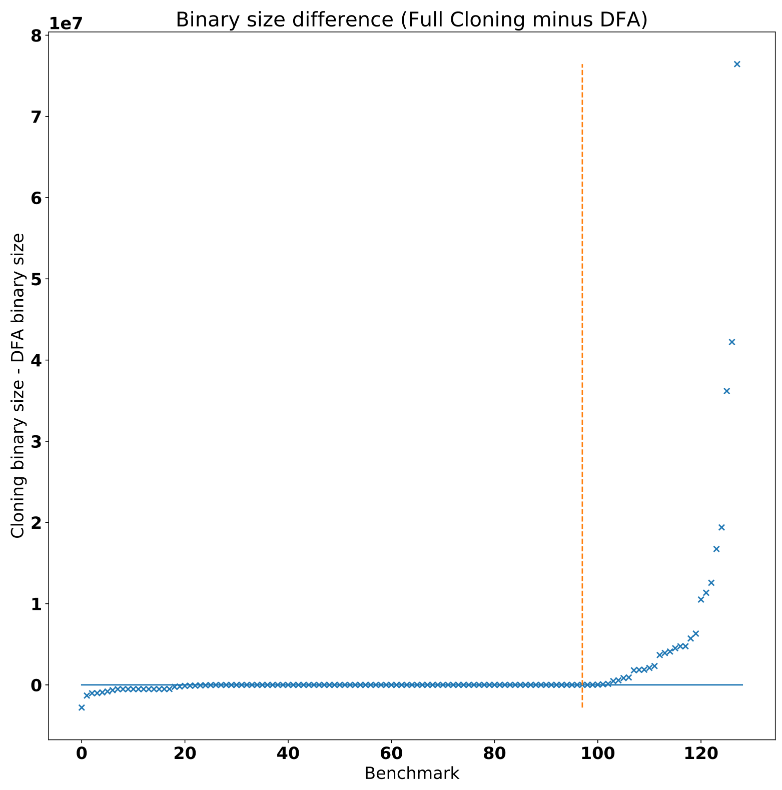


Figure 4.1. Size comparison (bytes) between the outline implementation of our approach (DFA) and full cloning (FC). The horizontal axis shows the 128 benchmarks in which we found optimization opportunities with context-sensitive constant propagation. The Y-axis shows the difference in bytes between the binary generated by FC and DFA (FC minus DFA). The vertical line shows the first benchmark in which DFA was better (positive y-value): cloning is better on 97 benchmarks, while outline DFA is better in 31.

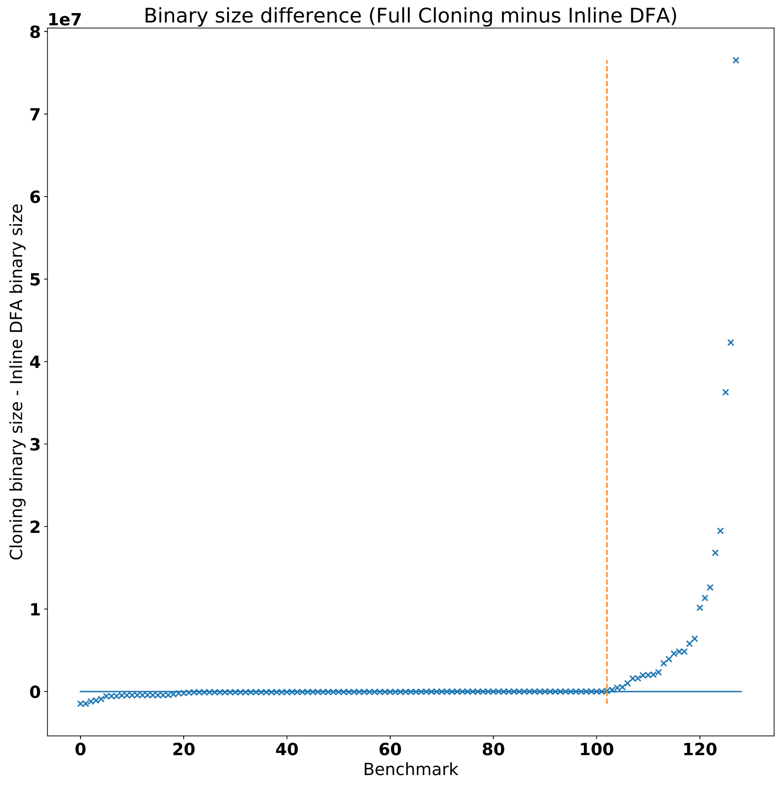


Figure 4.2. Size comparison (bytes) between the inline implementation of our approach (DFA) and full cloning (FC). The axes have the same meaning as in Figure 4.1 Cloning is better on 102 benchmarks, while inline DFA is better in 26.

faster than the binaries produced with the outline state machines (0.1% faster if we compare against the inline DFA). Once we analyze individual benchmarks, a few larger differences surface. For instance, McCat/05-eks yields binaries that are 43% faster with our outline approach than with full cloning. On the other hand, in Halide/blur we find the opposite behavior: full cloning leads to code that is 32% faster. These are, however, benchmarks which run for less than half a second, and thus the absolute differences are small. On the benchmarks which take at least one second to run, we did not observe any difference greater than 2%. In spite of these individual discrepancies, the experiment lets us conclude that the overhead of our state-machines and the dynamic dispatch of specialized code is negligible when compared to more traditional cloning-based approaches. The inline DFA was slightly faster than the outline implementation. However, the overhead introduced by both approaches was negligible in practice, giving a positive answer to our second research question.

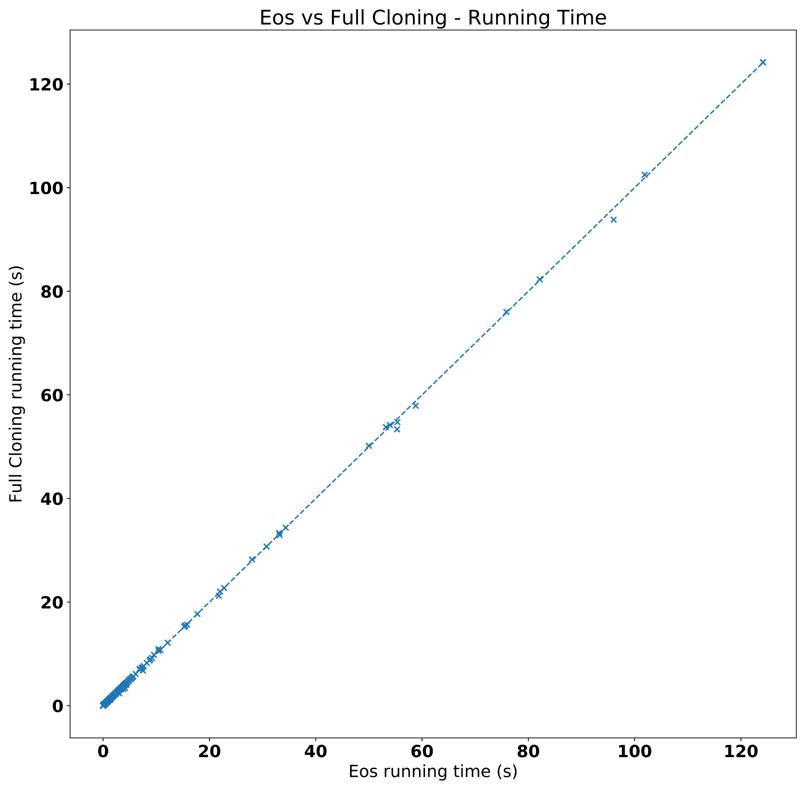


Figure 4.3. Comparison of the running time between the binaries generated by our approach (DFA) and full cloning (FC). The X axis shows the running time of the DFA binary in seconds; the Y axis shows the same for the cloning binary.

Figure 4.4 shows the ratio between the running time of the binaries output by outline DFA-based code generation and those generated by LLVM -O3 without our optimization. Values greater than 1 in the Y axis represent speed-ups, while values lesser than 1 occur for benchmarks in which our context-sensitive constant propagation made the program slower. Only the 128 benchmarks in which we found optimization opportunities are shown. Out of these, our approach generates speed-ups of over 2% in 15 benchmarks, and slowdowns of at least 2% in 18 programs. For two programs in FreeBench, `mason` and `fourinarow`, the speed-up we observed was slightly above 25%. The two benchmarks are cases in which function cloning generates a binary that is expressively smaller. As we explain in Section 4.5, this is due to its better interaction with dead code elimination. However, even if most of the optimized contexts are actually unreachable, there are important opportunities for context-sensitive optimizations in these programs that are not captured by LLVM -O3.

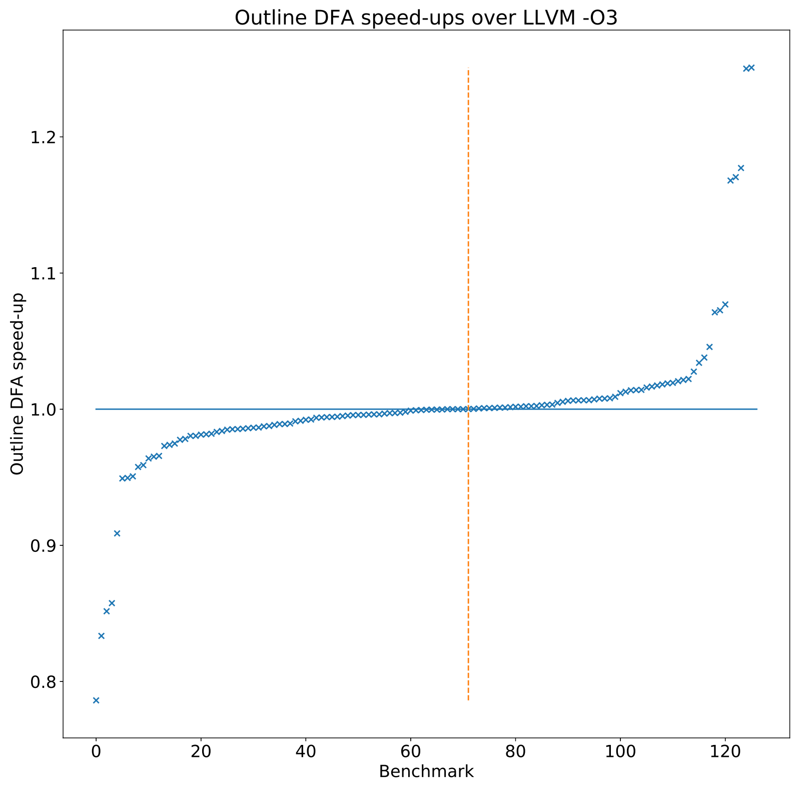


Figure 4.4. Speed-ups obtained by context-sensitive constant propagation implemented with the outline DFA method on top of LLVM -O3 without our optimization. The vertical line shows the first program for which its average running time was smaller when optimized by our pass. This happened in 55 programs (43% of the 128 benchmarks in which we found optimization opportunities).

Our implementation of context-sensitive constant propagation carelessly specializes code for new contexts whenever the gathered information allows it to do so. Hall [1991] has shown that unrestricted use of context-sensitive information usually does not lead to improvements, due to the performance penalty that comes when binary size grows, because of the lost cache-friendliness. Indeed, as we mentioned before, when we add the running time of all benchmarks, none of the implementations of our optimization achieves speed-ups. However, we wanted to demonstrate that (i) it is possible to carry out clone-based code specialization in large code bases with minimal overhead, and (ii) there are opportunities for doing so with performance gains. We have chosen constant propagation to perform this demonstration because it is difficult to conceive any optimization that could be more extensively applied than it.

Benchmark	B. (BL)	T. (BL)	B. (DFA)	T. (DFA)	B. (IDFA)	T. (IDFA)	B. (FC)	T. (FC)
lbm	18.16KB	33.17s	18.16KB	33.12s	22.17KB	33.24s	18.16KB	33.35s
mcf	18.18KB	17.85s	18.18KB	17.73s	26.18KB	17.96s	18.18KB	17.70s
libquantum	34.19KB	1.56s	98.19KB	1.51s	110.19KB	1.48s	74.19KB	1.50s
astar	38.38KB	94.45s	54.38KB	96.05s	78.38KB	96.31s	42.38KB	93.83s
bzip2	86.19KB	49.80s	110.19KB	49.99s	166.19KB	50.07s	134.19KB	50.18s
milc	98.70KB	15.27s	254.73KB	15.27s	290.73KB	15.36s	162.73KB	15.23s
sjeng	126.41KB	123.07s	2.84MB	124.13s	638.41KB	124.36s	178.41KB	124.21s
hmmer	168.73KB	53.73s	492.73KB	53.95s	604.73KB	53.78s	292.73KB	54.18s
namd	234.43KB	12.14s	234.43KB	12.15s	234.43KB	12.15s	234.43KB	12.17s
soplex	326.66KB	5.32s	326.66KB	5.34s	326.66KB	5.36s	326.66KB	5.33s
omnetpp	563.47KB	53.40s	563.47KB	55.30s	563.47KB	53.46s	563.47KB	53.36s
h264ref	623.62KB	75.85s	739.62KB	75.87s	751.62KB	75.41s	691.62KB	76.02s
povray	1.06MB	6.98s	1.06MB	6.89s	1.06MB	6.89s	1.06MB	6.95s
perlbench	1.18MB	20.99s	1.81MB	21.78s	2.59MB	21.08s	1.18MB	21.21s
xalancbmk	3.20MB	54.28s	3.20MB	55.37s	3.21MB	54.72s	3.20MB	54.75s
gobmk	3.29MB	101.94s	4.07MB	101.89s	4.23MB	102.04s	7.98MB	102.48s
gcc	3.42MB	0.93s	4.62MB	0.93s	4.62MB	0.93s	15.43MB	0.94s
sphinx3	77.78MB	7.57s	78.49MB	7.63s	78.47MB	7.55s	80.70MB	7.65s

Table 4.2. Results obtained on SPEC CPU2006 with baseline LLVM without our optimization (BL), and with context-sensitive constant propagation implemented with the outline DFA (DFA), inline DFA (IDFA) and full cloning (FC). Running times are the average of 5 runs. Each of the benchmarks have had their binary size measured (columns starting with "B."), as well as their running time (columns starting with "T.") for each technique.

4.4 Results on real-world programs

Table 4.2 shows results obtained on the CPU2006 benchmarks listed in Table 4.1. Benchmarks are sorted by their original binary size (generated by LLVM -O3). In 8 of the benchmarks (`astar`, `libquantum`, `xalancbmk`, `perlbench`, `sjeng`, `milc`, `h264ref`, `hmmer`) the outline DFA method generated a larger binary than cloning. Some cases are the result of better interaction between cloning and dead code elimination, when large portions of the optimizations computed by our context-sensitive constant propagation are on dead paths. This is the case in `perlbench`, for instance, where almost all of our optimizations are removed from the FC binary (many are also removed from the outline DFA binary). This kind of case will be better explored in the case study in Section 4.5.1. The other cases in which cloning performed better were, however, restricted to smaller benchmarks, in which absolute differences were mostly small (from 4KB in `xalancbmk` to 204KB in `hmmer`). On the other hand, the DFA outperforms cloning in 4 benchmarks (`gobmk`, `gcc`, `bzip2` and `sphinx3`). The 3 largest benchmarks in CPU2006 are all included in this set. In them, the DFA techniques are highly profitable in terms of binary size. For instance, the executable generated by any of the DFA-based methods is more than 3 times smaller than the output of full cloning in `gcc`, where the DFA saves almost 11MB. These benchmarks amount to the total space saving observed: when we sum all binary sizes from CPU2006, we see that the outline DFA adds 3 times less bytes to the original executable size in order to implement the same optimizations as cloning.

4.5 Case studies

In this section, we analyze the results of applying constant propagation on some specific benchmarks to help shed some light on how do cloning and our DFA-based method, with its two alternative implementations, behave in practice. These benchmarks were chosen based on the results obtained in them. We chose programs in order to illustrate each kind of outcome we have observed: when the DFA-based method was superior to cloning, when cloning was more profitable, and when each DFA implementation were better than each other.

4.5.1 TSVC and Four-in-a-row: when cloning enables dead code elimination

TSVC. The TSVC benchmarks[2] are a family of programs that consist of the same source code compiled with different values for global constants that control the behavior of the program. The functions/loops in TSVC are taken from a David Callahan's test suite for vectorizing compilers Callahan et al. [1988]. The results of all computations are not used - rather, "There is a dummy function called in each loop to make all computations appear required" (quoted from a comment in the source code). When applying context-sensitive constant propagation, generating explicit function clones makes it possible for the compiler to later realize the dummy function calls do not have any side effects, and they do not generate a result either. Thus, around 70% of the code is removed from the binary by later LLVM passes. The same happens with the outline implementation of the DFA-based method. These two algorithms, thus, make the output binary smaller in this benchmark.

TSVC is the only benchmark in which applying any of the tested methods caused the output binary size to decrease. On the other hand, applying the inline DFA-based method does not have the same outcome: the binary size increases by 10% – a more usual result. TSVC shows a case in which context-sensitive optimizations can decrease

[2]In the LLVM test suite, TSVC benchmarks can be found under `MultiSource/Benchmarks/TSVC`

code size by enabling more aggressive dead code elimination. That case seems artificial, however, as real world applications are unlikely to have large portions of the code that never execute even though there are static paths that reach them (in case there are no such paths, traditional approaches would already eliminate the code from the binary, without needing context-sensitive information). This is validated by the fact that none of the other benchmarks trigger this result.

Four in a row. Another case in which cloning enables better dead code elimination is in FreeBench's "Four in a row" benchmark, a program that implements an AI for the game with the same name. The AI uses the Minimax search algorithm with alpha-beta pruning. Minimax is a recursive algorithm in which one player tries to maximize its own score given the possible game states that it can reach in one move, while the second player tries to minimize the first player's score. The state tree is evaluated recursively using these two strategies, so that to decide each player's next move.

In this benchmark, our context-sensitive constant propagation algorithm dives deep in optimizing static paths that aren't reachable during execution. This does not happen because there is dead code in the original program; rather, such paths depend on a sequence of values for the parameters in the recursive calls that is not realizable. When the optimization is implemented with cloning, LLVM can easily see in a later stage that some function calls never occur, and then eliminate most of the optimized functions that would only be reached through those calls. On the other hand, the calls to "transition" or the switch statement in inline DFA hide this fact under another layer, making LLVM unable to do the same. In both cases, all the eliminated code is contained in optimized functions. Thus, unlike in TSVC, the binary still grows when the optimization is applied. But because most of the optimized code is dead, the binary generated with cloning is much smaller. The outline DFA method adds 528KB to the final binary. The inline DFA method implements the same optimization with 557KB of additional code. On the other hand, cloning only adds 4KB to the final binary size, since most of the generated intermediate code is discarded as dead.

This same behavior happens in a few other benchmarks. In general, clone-based optimizations interact better with other compiler optimizations because of the direct calls. When using the DFA-based methods, the compiler must deal with function pointers, which are harder to analyze.

4.5.2 When cloning is more space-efficient

The advantage of the DFA-based over cloning-based method consists of needing at most a single clone per function for tracking contexts regardless of how many times each function appears in the Optimization Tree. Such advantage, however, vanishes when few unoptimized functions appear multiple times in the Optimization Tree. When a function appears only once in the tree, both cloning and the DFA-based method will generate one clone of it. In addition to that, the DFA-based method will generate code, either in the `transition` function (outline implementation) or in the function itself (inline implementation) in order to track contexts – an additional overhead. In this case, cloning is usually more space-efficient. This is the case in SPECCPU2006's `473.astar` benchmark, a "path-finding library for 2D maps, including the well known A* algorithm." [3]. Our context-sensitive constant propagation optimization builds an Optimization Tree of 220 nodes for this program. Out of those, 139 ($\approx$ 63%) are optimized nodes - clones that can be avoided neither by cloning nor by DFA-based methods. Out of the remaining nodes, more than 80% consist of functions that appear only once in the tree. Therefore, the context-tracking DFA does not yield a significant reduction of the number of clones needed to implement the optimization. In this benchmark, the binary generated without our optimization has 39KB. DFA-based methods add 16KB (+41%) and 40KB (+104%) to that in the outline and inline implementations, respectively. On the other hand, cloning implements the same optimization adding just 4KB (+10%) to the binary. The same happens in many other benchmarks. However, was not observed in the largest benchmarks. When the Optimization Tree was larger, we were more likely to observe a larger fraction of repeated functions, making the DFA-based method more attractive.

4.5.3 When context-tracking with the DFA makes sense

In contrast to Section 4.5.2, on large Optimization Trees consisting of relatively few unique functions, encoding contexts using the DFA becomes more advantageous. This happens to be the case in most of the large benchmarks we experimented with, especially in SPEC. For example, in `403.gcc`, which consists of version 3.2 of the GNU C Compiler in a setup adapter for benchmarking, the original binary had 3.4MB. The applied optimization tree had 14,543 nodes, of which only about 39% consisted of optimized nodes – function copies that cannot be avoided. The binary optimized with cloning had 15.4MB – a 350% growth. Conversely, both the inline and outline DFA yielded a 4.6MB optimized binary (only 35% added to its size, meaning it needed 10 times less extra space). The prevalence of this case in the largest benchmarks adds up to the total result. The sum of the sizes of all original binaries was 107.9MB, compared to 141.27MB with the outline DFA method (+30.9%), 144.5MB with the inline DFA

[3]`https://www.spec.org/cpu2006/CINT2006/`

method (+33.9%), and 389.9MB with cloning (+261.2%). Even with this discrepancy, the outline DFA method was more space-efficient than cloning only in 31 out of the 95 benchmarks that were effectively optimized. However, because the DFA tends to be more compact in larger cases, the total result favors using it over cloning: it uses only 33.4MB of additional code in all binaries, around 8.5x less than cloning – which added 281.9MB.

While the inline DFA was slightly less space efficient than the outline implementation when summing the binary sizes of all benchmarks (144.5MB vs 141.3MB), the result is different when we restrict ourselves to the SPEC CPU2006 benchmarks (which are larger, real-world programs). Originally, the SPEC binaries generated by LLVM -O3 have a total size of 92.2MB. The binaries output by the inline DFA implementation add up to 97.9 (+6.15%), whereas the outline implementation generates slightly larger binaries (98.9, +7.27%). While the two results were close, they indicate that the inline DFA implementation tends to be at least as, or even more efficient than the outline implementation in large real-world programs.

Chapter 5

Conclusion

In this dissertation, we attempted to make context-sensitive optimizations more practical by reducing the code size penalty previously necessary to implement them. Previous approaches, namely cloning and inlining, when used in their classical form, need to copy the code of the full optimization paths in order to specialize a procedure for a given calling context. We proposed a method that mitigates this issue, requiring only a single clone of a function even if it appears multiple times in disjoint optimization paths. Instead of using clones to keep track of the calling context, we use an implicit state machine that is built statically and updated at run-time.

Our LLVM implementation was able to achieve significant gains over function cloning in terms of the additional space in the binary needed to implement context-sensitive constant propagation in 209 benchmarks. In SPEC CPU2006, a benchmark suite consisting of large real-world programs, our method needs 3 times less additional space than function cloning. If we consider all benchmarks, that number goes up to 8.5. On the other hand, the running time of the programs was practically unaffected: we maintain most of the gains of using classical clone-based specialization.

Making context-sensitive optimizations practical requires effort in two fronts: (i) efficiently deciding which calling contexts should be optimized and in which way, and (ii) generating compact code that implements the chosen specializations. We have advanced the second front. However, we have noted that the first one remains largely unexplored by previous work. Hall [1991] was the last work we are aware of to make extensive use of cloning-based context-sensitive specialization. Since then, context-sensitive analyses have become efficient enough to be used in industrial compilers (Lattner et al. [2007]). However, to the best of our knowledge, there are very few, if any, recent efforts to utilize the information generated by such analyses in context-sensitive optimizations. We have demonstrated the effectiveness of DFA-based code generation using a very naïve version of context-sensitive constant propagation. Contrary to what is suggested by Hall [1991], for example, our algorithm did not try to predict whether specializing a context would be profitable. Rather, it chose to propagate constants whenever it found opportunities to do so, until reaching a fixed limit of optimized contexts. Even though our implementation revealed optimization opportunities in the benchmarks we used, we believe there is vast room for improvement on the optimization itself.

Moreover, while we used constant propagation in our experiments, many other important optimizations could benefit from context-sensitive specialization. Notable examples include automatic parallelization, which is heavily dependent on pointer aliasing information. Currently, if in one calling context the pointer parameters of a function may alias, this optimization is hindered for all contexts. By using context-sensitive specialization, many more optimization opportunities could potentially be exploited.

Because few previous works propose optimizations that fit the framework we presented, we believe the mainstream adoption of our method is, in the short term, unlikely. DFA-based code generation still needs to be used to implement more aggressive optimizations, such as automatic parallelization, in order to gather attention. Developing those optimizations may prove to be a harder problem than the generation of compact code itself. Nevertheless, we believe the evidence we have gathered with experiments in real-world programs supports our research hypothesis: the implementation of context-sensitive code specialization is now more practical to perform with reasonable increases in binary size.

Bibliography

Ahn, W., Choi, J., Shull, T., Garzarán, M. J., and Torrellas, J. (2014). Improving javascript performance by deconstructing the type system. In *ACM SIGPLAN Notices*, volume 49, pages 496--507. ACM.

Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., and Pereira, F. M. Q. a. (2015). Runtime pointer disambiguation. In *OOPSLA*, pages 589--606, New York, NY, USA. ACM.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259--269, New York, NY, USA. ACM.

Ausiello, G., Demetrescu, C., Finocchi, I., and Firmani, D. (2012). k-calling context profiling. In *OOPSLA*, pages 867--878, New York, NY, USA. ACM.

Ball, T. and Larus, J. R. (1996). Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46--57. IEEE Computer Society.

Bond, M. D. and McKinley, K. S. (2007). Probabilistic calling context. In *ACM SIGPLAN Notices*, volume 42, pages 97--112. ACM.

Callahan, D., Dongarra, J., and Levine, D. (1988). Vectorizing compilers: A test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 98--105, Los Alamitos, CA, USA. IEEE Computer Society Press.

Cavazos, J. and O'Boyle, M. F. (2005). Automatic tuning of inlining heuristics. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 14. IEEE Computer Society.

Cooper, K. D., Hall, M. W., and Kennedy, K. (1993). A methodology for procedure cloning. *Comput. Lang.*, 19(2):105--117.

Das, D. (2003). Function inlining versus function cloning. *ACM SIGPLAN Notices*, 38(6):23--29.

Dean, J., Grove, D., and Chambers, C. (1995). Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77--101, London, UK, UK. Springer-Verlag.

Dragos, I. and Odersky, M. (2009). Compiling generics through user-directed type specialization. In *ICOOOLPS*, pages 42--47, New York, NY, USA. ACM.

Emami, M., Ghiya, R., and Hendren, L. J. (1994). Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242--256, New York, NY, USA. ACM.

Fähndrich, M., Rehof, J., and Das, M. (2000). Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, pages 253--263, New York, NY, USA. ACM.

Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*, pages 576--587, New York, NY, USA. ACM.

Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M. (2009). Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465--478, New York, NY, USA. ACM.

Ghiya, R. and Hendren, L. J. (1996). Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL*, pages 1--15, New York, NY, USA. ACM.

Grune, D., van Reeuwijk, K., Jacobs, H. E. B. C. J. H., and Langendoen, K. (2012). *Modern Compiler Design*. Springer, London, UK, UK, 2nd edition.

Hackett, B. and Guo, S.-y. (2012). Fast and precise hybrid type inference for javascript. In *PLDI*, pages 239--250, New York, NY, USA. ACM.

Hall, M. W. (1991). *Managing interprocedural optimization*. PhD thesis, Rice University, Houston, TX, USA. UMI Order No. GAX91-36029.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1--17.

Hind, M., Burke, M., Carini, P., and Choi, J.-D. (1999). Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848--894.

Hölzle, U., Chambers, C., and Ungar, D. (1991). Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, pages 21--38, London, UK, UK. Springer-Verlag.

Huang, J. and Bond, M. D. (2013). *Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management*, volume 48. ACM.

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.

Khedker, U., Sanyal, A., and Karkare, B. (2009). *Data Flow Analysis: Theory and Practice.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition. ISBN 0849328802, 9780849328800.

Korobeynikov, A. (2007). Improving switch lowering for the LLVM compiler system. In *SYRCoSE*, pages A.I--A.V, Innopolis, Russia. RAS.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, Washington DC. IEEE.

Lattner, C., Lenharth, A., and Adve, V. (2007). Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278--289, New York, NY, USA. ACM.

Lhoták, O. and Hendren, L. (2006). Context-sensitive points-to analysis: Is it worth it? In *CC*, pages 47--64, Berlin, Heidelberg. Springer.

Li, L., Cifuentes, C., and Keynes, N. (2013). Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM*, pages 85--96, New York, NY, USA. ACM.

Maydan, D. E., Hennessy, J. L., and Lam, M. S. (1991). Efficient and exact data dependence analysis. In *ACM SIGPLAN Notices*, volume 26, pages 1--14. ACM.

Metzger, R. and Stroud, S. (1993). Interprocedural constant propagation: an empirical study. *ACM Lett. Program. Lang. Syst.*, 2(1-4):213--232.

Might, M., Smaragdakis, Y., and Van Horn, D. (2010). Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI*, pages 305--315, New York, NY, USA. ACM.

Milanova, A. (2007). Light context-sensitive points-to analysis for java. In *PASTE*, pages 25--30, New York, NY, USA. ACM.

Milanova, A., Huang, W., and Dong, Y. (2014). CFL-reachability and context-sensitive integrity types. In *PPPJ*, pages 99--109, New York, NY, USA. ACM.

Milanova, A., Rountev, A., and Ryder, B. G. (2004). Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11(1):7--26.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA. ISBN 3540654100.

Oh, H., Lee, W., Heo, K., Yang, H., and Yi, K. (2014). Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, pages 475--484, New York, NY, USA. ACM.

Petrashko, D., Ureche, V., Lhoták, O., and Odersky, M. (2016). Call graphs for languages with parametric polymorphism. In *OOPSLA*, pages 394--409, New York, NY, USA. ACM.

Reps, T. (1997). Program analysis via graph reachability. In *ILPS*, pages 5--19, Cambridge, MA, USA. MIT Press.

Reps, T. (2000). Undecidability of context-sensitive data-dependence analysis. *TOPLAS*, 22(1):162--186.

Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49--61, New York, NY, USA. ACM.

Rus, S., Rauchwerger, L., and Hoeflinger, J. (2002). Hybrid analysis: Static & dynamic memory reference analysis. In *ICS*, pages 274--284, New York, NY, USA. ACM.

Sallenave, O. and Ducournau, R. (2012). Lightweight generics in embedded systems through static analysis. In *LCTES*, pages 11--20, New York, NY, USA. ACM.

Samadi, M., Hormati, A., Mehrara, M., Lee, J., and Mahlke, S. (2012). Adaptive input-aware compilation for graphics engines. In *PLDI*, pages 13--22, New York, NY, USA. ACM.

Santos, H. N., Alves, P., Costa, I., and Quintao Pereira, F. M. (2013). Just-in-time value specialization. In *CGO*, pages 1--11, Washington, DC, USA. IEEE Computer Society.

Sarimbekov, A., Sewe, A., Binder, W., Moret, P., Schoeberl, M., and Mezini, M. (2011). Portable and accurate collection of calling-context-sensitive bytecode metrics for the java virtual machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 11--20. ACM.

Shivers, O. (1988). Control flow analysis in scheme. In *PLDI*, pages 164--174, New York, NY, USA. ACM.

Sperle Campos, V. H., Alves, P. R., Nazaré Santos, H., and Quintão Pereira, F. M. (2016). Restrictification of function arguments. In *CC*, pages 163--173, New York, NY, USA. ACM.

Spivey, J. M. (2004). Fast, accurate call graph profiling. *Software: Practice and Experience*, 34(3):249--264.

Sridharan, M. and Bodík, R. (2006). Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387--400, New York, NY, USA. ACM.

Stucki, N. and Ureche, V. (2013). Bridging islands of specialized code using macros and reified types. In *SCALA*, pages 10:1--10:4, 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, New York, NY, USA. ACM.

Wang, H., Wu, P., and Padua, D. (2014). Optimizing r vm: Allocation removal and path length reduction via interpreter-level specialization. In *CGO*, pages 295:295--295:305, New York, NY, USA. ACM.

Wei, S. and Ryder, B. G. (2015). Adaptive context-sensitive analysis for javascript. In *ECOOP*, pages 712--734, London, UK, UK. Springer.

Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131--144, New York, NY, USA. ACM.

Wilson, R. P. and Lam, M. S. (1995). Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1--12, New York, NY, USA. ACM.

Zhu, J. and Calman, S. (2004). Symbolic pointer analysis revisited. In *PLDI*, pages 145--157, New York, NY, USA. ACM.