

Dynamic Dispatch of Context-Sensitive Optimizations

GABRIEL POESIA, Stanford University, USA

FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

Academia has spent much effort into making context-sensitive *analyses* practical, with great profit. However, the implementation of context-sensitive *optimizations*, in contrast to analyses, is still not practical, due to code-size explosion. This growth happens because current technology requires the cloning of full paths in the Calling Context Tree. In this paper, we present a solution to this problem. We combine finite state machines and dynamic dispatching to allow fully context-sensitive specialization while cloning only functions that are effectively optimized. This technique makes it possible to apply very liberal optimizations, such as context-sensitive constant propagation, in large programs—something that could not have been easily done before. We demonstrate the viability of our idea by formalizing it in Prolog, and implementing it in LLVM. As a proof of concept, we have used our state machines to implement context-sensitive constant propagation in LLVM. The binaries produced by traditional full cloning are 2.63 times larger than the binaries that we generate with our state machines. When applied on Mozilla Firefox, our optimization increases binary size from 7.2MB to 9.2MB. Full cloning, in contrast, yields a binary of 34MB.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Procedures, functions and subroutines*; • **Theory of computation** → *Regular languages; Operational semantics*.

Additional Key Words and Phrases: Compiler, Context-sensitive optimization, Dynamic dispatch

ACM Reference Format:

Gabriel Poesia and Fernando Magno Quintão Pereira. 2020. Dynamic Dispatch of Context-Sensitive Optimizations. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 167 (November 2020), 28 pages. <https://doi.org/10.1145/3428235>

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, academia and 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; Jeong et al. 2017; Li et al. 2020; Might et al. 2010; Milanova 2007; Milanova et al. 2014; Oh et al. 2014; Späth et al. 2019, 2016; Thakur and Nandivada 2019, 2020; Thiessen and Lhoták 2017; Wei and Ryder 2015; Wilson and Lam 1995; Yu et al. 2010]. Presently, we believe that 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 seems still an unsolved problem.

Authors' addresses: Gabriel Poesia, Computer Science, Stanford University, USA, me@gpoesia.com; Fernando Magno Quintão Pereira, Computer Science, UFMG, Brazil, fernando@dcc.ufmg.br.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/11-ART167

<https://doi.org/10.1145/3428235>

Implementing Context-Sensitive Optimizations is Challenging. Applying the results of context-sensitive optimizations is still difficult. The usual approach towards this end is to perform some optimization once the context-sensitive analysis shows that said 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, on the contexts where it is safe to do so. This is accomplished via function inlining or cloning [Cooper et al. 1993; Das 2003; Dean et al. 1995a; Hall 1991; Metzger and Stroud 1993; Petrashko et al. 2016]. Nevertheless, due to code size expansion, compilers do not resort to this kind of specialization, or apply it in limited ways. One of the key challenges preventing specialization is the fact that not only the optimized functions must be cloned, but whole paths of procedures within the program’s call graph must be also replicated. 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 2.

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 [Fluckiger et al. 2020]. In this case, context-sensitive information is readily available, because the runtime engine has access to the state of the stack once it moves from interpretation to compilation. Examples of code specialization in the JIT world include constant propagation [Leopoldseder et al. 2018; Santos et al. 2013], type speculation [Dean et al. 1995a; Gal et al. 2009; Hackett and Guo 2012], method resolution via inline caches [Hölzle et al. 1991; Vergu and Visser 2018], fast symbol lookups [Stadler et al. 2016] and operation specialization [Vergu and Visser 2018; Wang et al. 2014]. In some implementations of just-in-time compilers code size explosion is not a problem: if a new version of a specialized function needs to be produced, its old binary can be discarded [Lima et al. 2020]. In other implementations [Inoue et al. 2011], code size explosion is still troublesome, often preventing more extensive optimizations. Similarly, static compilers perform specialization in restricted scenarios. For instance, gcc, at the -O2 optimization level, creates specialized versions of a function if it can infer that some of its parameters are constant only in specific contexts, and it deems the tradeoff between speedup and code size increase to be worthwhile¹. DawnCC² restricts specialization to two levels of the function stack [Poesia et al. 2017]. Similarly, specialization of generic functions is common in languages with parametric polymorphism [Petrashko et al. 2016]. Finally, there exists a whole branch of partial evaluation dedicated to code specialization [Brown and Palsberg 2017]. In this paper, we present a technique that makes this sort of context-sensitive specialization practical in general for static compilers.

Contribution: Mitigating Code Explosion with State Machines. 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 on 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 Li et al. [2013] 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

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

²<http://cuda.dcc.ufmg.br/dawn/>

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

Summary of Results. To validate our ideas, we have implemented a context-sensitive version of inter-procedural 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, are equivalent to unrestricted inlining: they clone every function in contexts leading to specialized procedures. We chose constant propagation as our example of context-sensitive code specialization because it is simple and extensive: it is hard to think about a static optimization that produces a larger number of specialized functions. Nevertheless, our ideas can be combined with other optimizations that rely on context-sensitive information and code cloning. We have tested our technique onto 209 benchmarks, taken from the LLVM test suite and from SPEC CPU2006/CPU2017. The executables that we produce are as fast as those produced with full cloning. The programs optimized with our approach took 1,517 seconds to run in total, vs 1,515 seconds taken by the programs transformed with full cloning. In other words, the overhead of our state machines is not statistically significant within a confidence interval of 95%. On the other hand, their benefit, in terms of code size reduction, is substantial. We generate 141.27MB for those 209 executables, vs 389.85MB produced with full cloning. We can summarize our contributions as follows:

Method: Our key contribution is 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 proportional to the size of the context call tree. As we explain in Section 2, the core idea behind this technique is a combination of state machines and indirect calls, *à la* dynamic dispatch, to select particular calling contexts. Our approach handles recursive programs, and can be parameterized to different compiler optimizations.

Formalization: Section 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 technique. That section 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.

Implementation: Section 4 describes the implementation of our technique in LLVM 10.0.0, 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. Although we have experimented only with context-sensitive constant propagation, our ideas can be combined with other optimizations that rely on context-sensitive information and code cloning.

2 OVERVIEW

The vast majority of computer languages provide developers with the abstraction of *functions* (or *procedures*, *subroutines*, *methods*, etc). If a dataflow analysis propagates information across the boundaries of functions, then it is called *interprocedural*. Interprocedural analyses are *context insensitive* or *context sensitive*. In this paper, we focus on the latter. For an informal definition of this family 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]

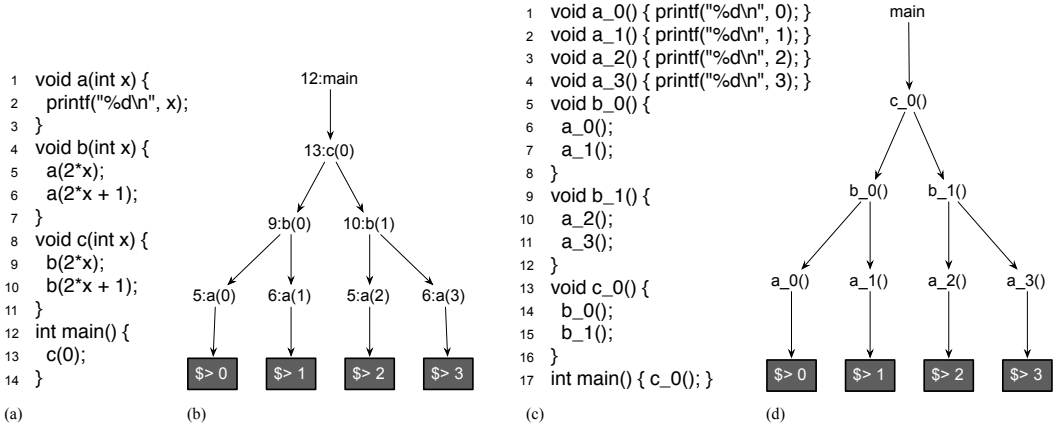


Fig. 1. (a) Program before context-sensitive optimization. (b) Calling context tree of original program. (c) Program after context-sensitive constant propagation. (d) CCT of the optimized program.

The *calling context* of an activation of function f is given by the sequence of function calls currently stacked when f is invoked. Definition 2.1 formalizes this notion. According to Definition 2.1, we can refer to the instructions in a program using some consistent numbering scheme, such as line numbers if we disallow having two statements on the same line.

Definition 2.1 (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 $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n$. The empty context, in which execution begins, is denoted by C_0 . The function invoked by C_n (the last call instruction of C) is the *active function* of C .

2.1 The Challenge of Implementing Context-Sensitive Optimizations

Context-sensitive analyses can be 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 on 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 [Reps 2000].

If the problem of obtaining context-sensitive information statically 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 and cloning. Many industrial compilers, such as gcc, LLVM, Open64, HotSpot and Mozilla’s IonMonkey implement inlining at higher optimization levels. Cloning is also found in mainstream products. For instance, gcc clones a function if it has only one calling site in the entire program, and its linkage is internal. A similar approach is adopted by Mozilla’s TraceMonkey, albeit dynamically [Santos et al. 2013]. As a final example, Scala clones generic functions that are marked with the @specialized annotation. The difficulty of applying either inlining or cloning stems from the same problem: code size explosion. The unrestricted application of any of these techniques might result in an optimized program that is exponentially larger than its original version. Example 2.2 illustrates 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 2.2. Figure 1 (a) shows a C/C++ program, plus its CCT, which contains 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 (c), and its calling context tree appears in Figure 1 (d). The graphs in Figures 1 (b) and (d) are isomorphic, as they encode the same semantics. 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 (a) varies at runtime.

To implement interprocedural context-sensitive constant propagation in the program of Figure 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*³, and the call sites leading to its invocation the *path*. This example takes us to one of the key shortcomings of 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 endeavour. Figure 2 provides some perspective on previous attempts, contrasting them with this work. The figure distinguishes the static analysis that enables an optimization, and the optimization itself. All those 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. [Petrashko et al. \[2016\]](#) only clones leaves, but, in this case, cloning happens when the static information, even though context-sensitive, is invariant at a given program point where a function is invoked, regardless of the calling path leading to it. 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, [Li et al. \[2013\]](#) only clone 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 2.2, our approach only clones leaves. We might transform functions in the calling path; nevertheless, their implementations shall remain unique.

	Analysis	Optimization
Petrashko et al. 2016	CS	CI
Full inlining	CS	CS (graph)
Li et al. 2013	CS	CS (path)
This paper	CS	CS (leaf)

Fig. 2. Previous work in perspective.

2.2 State Machines to the Rescue

To circumvent the problem just mentioned, 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 2.3, and illustrated in Example 2.4.

Definition 2.3 (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

³Notice that leaf functions might call other routines—this terminology only indicates that the function is optimizable.

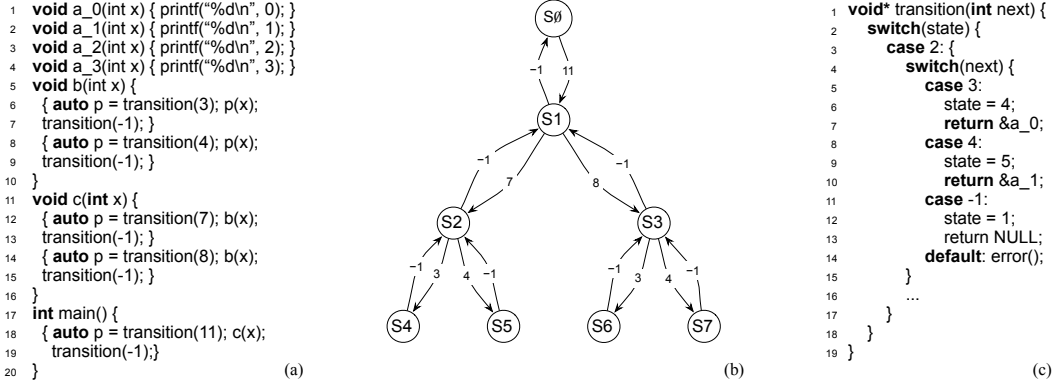


Fig. 3. (a) Program after context-sensitive optimization. (b) State machine that tracks context changes. (c) The transition function, which implements state switching and selects which version of an optimized function to call.

allow O to specialize F in context C , i.e., $O(F, C) = F'$, $F \neq F'$. Hence, leaf functions are amenable to optimization in some contexts. To implement O , the optimization proposed in this paper delivers the following guarantee: it produces one clone of F for each $A(F, C)$ that yields non-trivial static facts. Thus, unoptimized functions are not duplicated.

Example 2.4. Figure 3 (a) shows an optimized version of the program earlier seen in Figure 1. The context-sensitive optimization is constant propagation. The calling context tree has four leaves, one for each activation of function a —each with a different constant as argument. Figure 3 (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 3 (b). At each state shift, $transition$ might return a pointer to the next function to be invoked. An example of the code that performs this action appears in Figure 3 (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 runtime, 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.

In Figure 3 (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 asymptotic worst-case bound on the number of clones: it is still exponential on the program size. To see how this worst case emerges, it suffices to consider a balanced binary calling context tree, in which every leaf represents an optimized function.

3 FORMAL SPECIFICATION

This section explains how we implement context-sensitive optimizations. In Section 3.1 we introduce MiniLog, a core programming language that gives us the necessary syntax to explain our ideas. We have implemented MiniLog in Prolog, and have used this implementation to design the algorithms that we present in this paper. Our code transformation engine is parameterized by a context-sensitive optimization, a notion defined in Section 3.2. Given such an optimization, Algorithm 1 (Section 3.3) 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 FSM is the subject of Section 3.4. Section 3.5 states a few properties of our approach. Section 3.6 provides details of our implementation in LLVM. Finally, Section 3.7 discusses an alternative implementation of the state transitions based on an inlined state machine. Proofs of Theorems and Properties presented in this Section are available as supplementary material stored in the ACM Digital Library.

3.1 MiniLog: core Syntax

Figure 5 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 in 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. Nevertheless, we emphasize that MiniLog's actual implementation, as well as all the developments in the rest of this paper, 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 4 (a) shows our original example, introduced in Figure 1 (b), implemented in MiniLog. The expression $2 * x + 1$ (originally in line 12 of Figure 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 5 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 4 (a).

Given MiniLog's syntax, Definition 3.2 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. Example 3.3 illustrates these observations.

Definition 3.2 (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 $\mathbf{program}(P)$ be a syntactically valid MiniLog program.

Example 3.3. Figure 4 (b) shows the calling context tree of the program that appears in Figure 4 (a). We have two contexts which are amenable to be optimized by constant propagation: $C_0 = c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$, and $C_1 = c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_4$. 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 $O(C_0) = \mathbf{function}(a_0, x, \mathbf{print}(0))$, and that $O(C_1) = \mathbf{function}(a_1, x, \mathbf{print}(1))$.

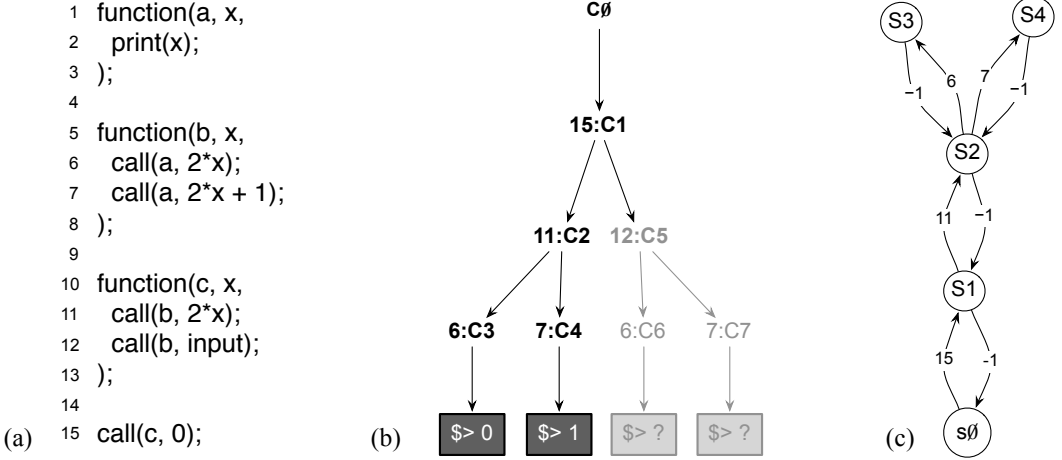


Fig. 4. (a) Modified version of program seen in Figure 1. The unknown variable input, in line 12, prevents constant propagation from optimizing the whole context tree. (b) Calling Context Tree of the program. We have highlighted the Optimization Tree (see Definition 3.6). (c) State Machine produced for this example. Each context in the optimization tree creates a new state.

Values:	$\frac{V \in \{\dots, -1, 0, 1, 2, \dots\}}{\text{exp}(V)}$	Names:	$\frac{A \in (a \dots zA \dots Z)^*}{\text{name}(A)}$	Variables:	$\frac{\text{name}(A)}{\text{exp}(A)}$
Instructions:	$\frac{\text{instruction}(I)}{\text{program}(I;)}$	Programs:	$\frac{\text{instruction}(I) \quad \text{program}(P)}{\text{program}(I; P)}$		
Output:	$\frac{\text{exp}(E)}{\text{instruction}(\text{print}(E))}$	Arithmetics:	$\frac{\text{exp}(E_1) \quad \text{exp}(E_2)}{\text{exp}(E_1 \oplus E_2), \oplus \in \{+, -, \times, \text{etc}\}}$		
Allocation:	$\frac{\text{name}(A) \quad \text{exp}(E)}{\text{instruction}(\text{set}(A, E))}$	Invocation:	$\frac{\text{name}(Name) \quad \text{name}(Arg) \quad \text{exp}(Ret)}{\text{instruction}(\text{call}(Name, Arg, Ret))}$		
Assignment:	$\frac{\text{name}(A) \quad \text{exp}(E)}{\text{instruction}(\text{def}(A, E))}$	Function:	$\frac{\text{name}(Name) \quad \text{name}(Arg) \quad \text{program}(Body)}{\text{instruction}(\text{function}(Name, Arg, Body))}$		
Return:	$\frac{\text{exp}(E)}{\text{instruction}(\text{return}(E))}$	Conditional:	$\frac{\text{exp}(E) \quad \text{program}(P_1) \quad \text{program}(P_2)}{\text{instruction}(\text{ifelse}(E, P_1, P_2))}$		

Fig. 5. The Syntax of MiniLog.

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 3.2). 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 3.4.

Definition 3.4 (Complete contexts). A context C is a *complete context* of a MiniLog program P if P_i , its first **call** instruction, is not contained in the body of any function declaration⁴.

Example 3.5. The context formed by the calls at lines 15, 11, 6, reached during the execution of the program in Figure 4 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 3.6, we discuss how we have adapted the presented solution to accommodate *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 3.6 (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 geometric 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 3.7 sheds some light on this definition.

Example 3.7. We have highlighted, in Figure 4 (b), the optimization tree that exists embedded in the calling context tree of the program in Figure 4 (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 3.8 summarizes this result.

THEOREM 3.8. *Let P be a MiniLog program. Let $C \rightarrow C'$ 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' , 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 $T(O)$.

3.3 Code Transformation

The cases in Theorem 3.8 are the basis of our method of tracking calling contexts at run-time. 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.

⁴This 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`.

Code generation is based on a Finite State Machine (FSM) derived from $T(O)$. In this section, we show how to transform the program assuming that we have the state machine. Section 3.4 shows how to build the FSM. Algorithm 1 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 3.4 shows how to build the FSM and implement transition. For now, the following assumptions on transition are made:
 - transition takes one parameter: the input argument to the transition function of the FSM. 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 is an optimization, to avoid 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 1 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 traditional function 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 3.9. Figure 6 shows the code that Algorithm 1 produces for the program in Figure 4 (a), using the optimization tree in Figure 4 (b). The state machine produced in this case appears in Figure 4 (c). We have produced two clones of function `a`: `a_0` and `a_1`; one per optimized context.

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 FSM. Such functions are produced by step (3) of Algorithm 1. We refer to these functions as *auxiliary path clones*.

Example 3.10. Figure 3.9 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 $C_1 \rightarrow C_5 \rightarrow C_6$ and $C_1 \rightarrow C_5 \rightarrow C_7$ (see Figure 4), which could not be optimized by

```

Data: Program  $P$ , Context-sensitive optimization  $O$ 
Result: Optimized program  $OptP$ 
/* (1) Declare FSM '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 1: Code generation procedure that populates the MiniLog program with calls to the FSM that tracks contexts.

constant propagation. Notice that, independent on the optimization tree, we would not produce more versions of `ctx_sens_b` and `ctx_sens_c`.

3.4 Construction of the Finite State Machine

The cases in Theorem 3.8 are our basis for building an FSM derived from $T(O)$. Algorithm 2 builds the FSM needed by Algorithm 1. In phase 1, every node of $T(O)$ becomes a state in the FSM. Phase 2 covers transitions that fall into Case 1 of Theorem 3.8. Call instructions that correspond to edges in $T(O)$ become transitions in the FSM, and their identifier in the program matches the corresponding transition's label in the FSM. Moreover, the FSM 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 3.8 is handled implicitly. Algorithms 1 and 2 are completely oblivious to transitions between contexts not in $T(O)$; hence, such calls are processed as in the original program.

Calls that force the program flow to leave the Optimization Tree (Case 2 in Theorem 3.8) 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 1; thus, the context-insensitive version of the callee is invoked. In other words, context-insensitive functions cannot provoke transitions in the FSM. Therefore, when a context-insensitive function returns, the FSM 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

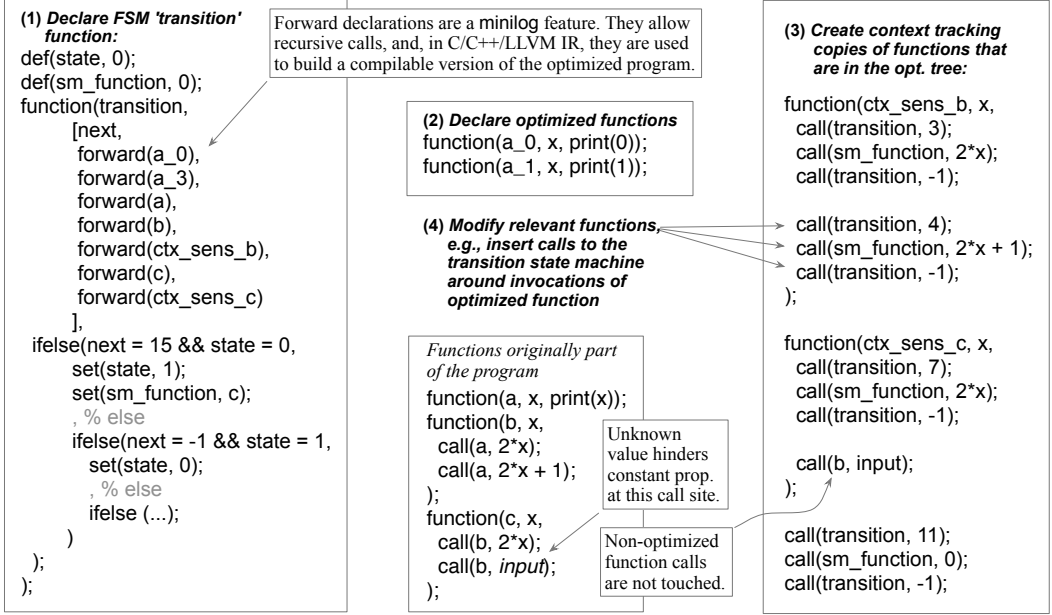


Fig. 6. Code produced by Algorithm 1, to implement constant propagation on the program seen in Figure 4 (a). The box on the right shows the optimized version of the code in the center-bottom part of the figure.

cause the program to leave $T(O)$ in some contexts, but transition inside $T(O)$ in others. Such calls require special care. Example 3.11 illustrates this concept.

Example 3.11. Figure 7 (a) shows the program earlier seen in Figure 1, written in MiniLog. Contrary to the MiniLog program in Figure 4, 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 $C_0 \rightarrow C_2 \rightarrow C_3$ and $C_0 \rightarrow C_5 \rightarrow C_7$ are optimized. The optimization tree appears in Figure 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 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 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 3.11, 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 FSM 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 FSM to return to the last state visited while the program was within the optimization tree. This scheme, implemented in Phase 3 of Algorithm 2, completes our coverage of all possible transition types in Theorem 3.8.

The transition function used by Algorithm 1 is implemented in MiniLog as a sequence of **ifelse** statements, that test all $(CurrentState, Input)$ pairs. In our LLVM implementation,

```

Data: Context-sensitive optimization  $O$ 
Result: Finite State Machine  $FSM$ 
 $FSM \leftarrow \mathbf{new}$  Finite State Machine;
/* (1) Create states for nodes in  $T(O)$ . */
 $OptTree \leftarrow T(O)$ ;
forall  $Node \in OptTree.Nodes$  do
  |  $FSM.addState(Node.Label)$ ;
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 */
 $TransitionSet = \mathbf{empty\ map} : Function \mapsto Set < Edge, label >$ ;
/* (2) Create transitions for calls/returns inside  $T(O)$  (Case 1 of Thm. 3.8). */
forall  $Node \in OptTree.Nodes$  do
  | forall  $Edge \in Node.EdgesToChildren$  do
    |  $FSM.addTransition(Node.Label, Edge.Destination.Label, Edge.CallInstLabel)$ ;
    |  $FSM.addTransition(Edge.Destination.Label, Node.Label, -1)$ ;
    |  $TransitionSet[Node.ActiveFun] \leftarrow TransitionSet[Node.ActiveFun] \cup \{Edge.CallInstLabel\}$ ;
  | end
end
/* (3) Create return-only states & transitions that leave  $T(O)$  (Case 2 of
Thm. 3.8). */
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  $MissingTransitions \neq \emptyset$  then
  | |  $ReturnOnlyStateLabel \leftarrow concatenate("R", Node.Label)$ ;
  | |  $FSM.addState(ReturnOnlyStateLabel)$ ;
  | |  $FSM.addTransition(ReturnOnlyStateLabel, Node.Label, -1)$ ;
  | | forall  $MissingLabel \in MissingTransitions$  do
  | | |  $FSM.addTransition(Node.Label, ReturnOnlyStateLabel, MissingLabel)$ ;
  | | end
  | end
end

```

Algorithm 2: Algorithm for building a Finite State Machine which tracks contexts that are relevant to a given context-sensitive optimization.

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 FSM after the transition is a node of $T(O)$, then either the optimized function (given by O , in Definition 3.2) 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.

THEOREM 3.12 (CORRECTNESS). *The transformation CS preserves the semantics of programs.*

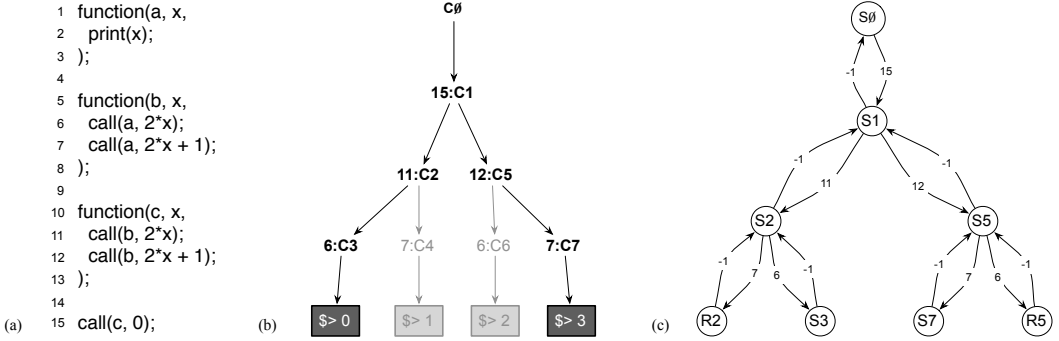


Fig. 7. (a) Example of program that can be fully optimized with constant propagation. (b) Optimization Tree that we obtain, assuming that only the two highlighted contexts are optimized. (c) The FSM derived from the Optimization Tree. In the FSM, a *return-only* state corresponding to state C_i is labeled R_i . These states are created by step (3) of Algorithm 2.

3.5 Properties

Given a context-sensitive optimization O to be applied onto a program P , we use Algorithm 2 to generate the FSM that supports the application of O , and use Algorithm 1 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 FSM. Property 3.13 puts a bound on the number of clones that we produce for a given optimization tree.

PROPERTY 3.13. *Algorithm 1 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.*

The generated transition function will contain a representation of the entire Optimization Tree; hence, the FSM’s size can be exponential. However, the FSM 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 Section 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 3.14 summarizes this fact.

PROPERTY 3.14. *Algorithm 2 produces an FSM with at most 2 states per node in $T(O)$*

3.6 From MiniLog to LLVM’s Intermediate Representation

We have implemented the ideas discussed in this section in LLVM, as an extension of the opt middle-end optimizer. This implementation follows Algorithms 1 and 2; however, to make it practical, we have adopted two extensions, which we describe in this section. The first extension allows us to support context-sensitive optimizations described by *partial contexts*. The second extension allows us to support exception handling in functions that have this feature.

Supporting partial contexts. Algorithms 1 and 2 assume that the root of an optimization tree is 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 contexts.

To benefit from this observation, we have extended 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 value, -1 , to indicate function return. However, the same approach would not work in a language that supports exceptions. To handle exceptions, we have adopted a slightly different approach. 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)` can be 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.

3.7 Inline DFA Implementation

Algorithm 1 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 3.4, each invocation of this function executes 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`).

However, it is possible to bring the cost per transition down to a single switch statement instead of two, and avoid the function calls altogether. To understand how this optimization works, suppose that we inlined the `transition` function everywhere where it is called with a parameter different than -1 , i.e. once in all call sites that may trigger a DFA transition. Consider one such call site CS , in the body of a function f . The second (nested) switch statement in the body of `transition` checks which call site we are at. However, since `transition` is inlined at CS , this check is unnecessary. Therefore, the innermost switch can be replaced by the body of its “ CS ” case; thus, yielding a single switch statement which tests for the DFA state.

Here comes the key observation that makes this approach practical: given that the call site CS is located in function f , not all states are possible. Rather, only the DFA states in which function f is active are possible—usually a small subset of all states. Thus, other states can be removed from the switch statement. We are still left with the call to `transition(-1)`. Nevertheless, this call can be eliminated, because it will always bring the DFA to the state active when the execution reaches

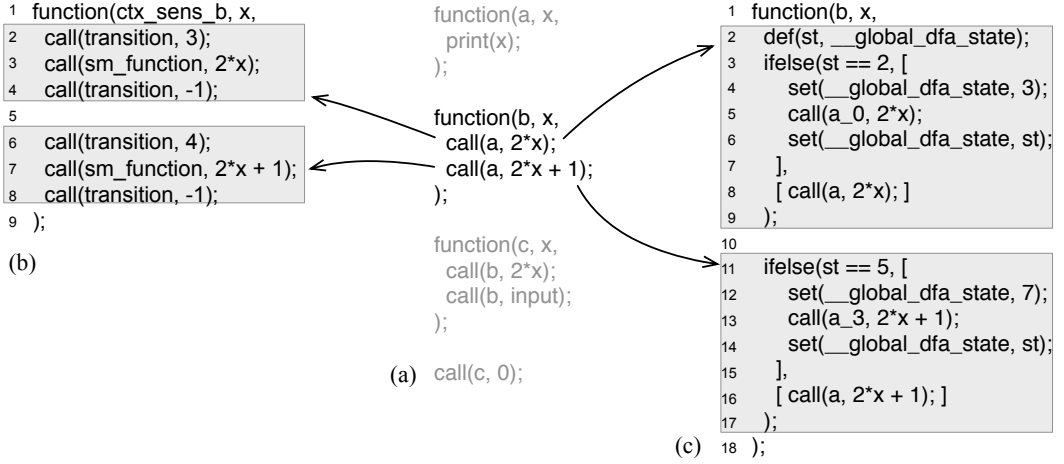


Fig. 8. (a-Middle) Program taken from Figure 7. (b-Left) Function b, optimized using the approach seen in Section 3.3. (c-Right) The same function optimized with the inlined DFA discussed in Section 3.7.

function f . Therefore, in the beginning of f we save the current DFA state in a local variable st . Then, calls to `transition(-1)` are replaced by a statement assigning the global state back to the state saved in st . This inline implementation usually needs more space than its outline counterpart; however, it avoids the cost of calling `transition`, and might enable intra-procedural optimizations.

Example 3.15. Figure 8 (c) shows function b from Figure 7 optimized with the inline DFA implementation described in this section. The applied Optimization Tree is also the one from Figure 7. Given the contexts shown in Figure 4, we know that if the program is currently at b, there are only two possible states: S_2 and S_5 . 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.

4 EVALUATION

To evaluate our context-sensitive code generation method, we have implemented it in LLVM 10.0.0. We run experiments on Intel i7-3770 at 3.40GHz, with 16GB of RAM featuring Ubuntu 16.04. We report results for our core technique augmented with the improvements discussed in Section 3.7. Our optimization runs as the first pass in LLVM’s Link-Time Optimization (LTO) pipeline; after every compilation unit has been optimized with `clang -O2`. We compare it with three baselines: `clang -O25`, `clang -O0` and `clang -O2` plus full cloning. In this section, we answer the following research questions:

RQ1 What is the overhead that our technique adds onto compilation time?

RQ2 What is the overhead that our technique adds onto the size of compiled code?

RQ3 What is the overhead that our approach adds onto the running time of compiled code?

RQ4 How does our approach scale when we change the number of contexts considered?

Context-Sensitive Constant Propagation. We evaluate our techniques over 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

⁵Results with `clang -O3` can be found at Poesia [2017]. They are similar to results with `clang -O2`, as both apply extensive function inlining. In contrast, at the `-O1` level, inlining is only applied to functions marked as `always-inline`.

Table 1. Benchmarks in SPEC CPU2017. **F.** and **Insts.** are the number of functions and instructions in the original program, respectively. **Opt. C.** 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. N.** gives us the number of nodes in the optimization tree. This number is the quantity of clones created by typical clone-based specialization. **DFA S.** shows the number of states in the DFA we use to dynamically dispatch optimizations.

Benchmark	F.	Insts.	Contexts	Opt. C.	Opt. N.	DFA S.
500.perlbench	1,941	434,745	$> 10^{16}$	6,426	14,998	15,000
502.gcc	10,405	1,619,027	$> 10^{16}$	18	14,994	29,954
505.mcf	27	6,443	166	10	15,000	15,002
508.namd	120	193,944	135	0	0	0
510.parest	3,715	512,550	36	2	2	3
511.povray	1,318	183,329	37	0	0	0
519.lbm	15	2,089	101	7	8	9
520.omnetpp	5,245	264,857	5	0	0	0
523.xalancbmk	10,491	749,843	111	13	21	22
526.blender	34,686	1,847,308	13,808,980,262	7,906	13,076	13,080
531.deepsjeng	86	15,310	124,147,605	12,388	14,857	14,860
538.imagick	988	214,061	$> 10^{16}$	11,424	14,998	15,000
541.leela	207	17,355	97	0	0	0
544.nab	76	21,007	94,746	168	14,575	14,577
557.xz	207	24,267	645	38	58	59
999.specrand	6	233	62	0	0	0

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 optimization 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, like in previous work [Hall 1991; Li et al. 2013]. We have chosen to implement context-sensitive constant propagation because it is simple and extensive. The goal of this implementation is to measure the overhead of our state-machines when compared to its version based on full function cloning. Being a research artifact, we do not expect it to speedup code produced with `clang -O2`, although, as we shall discuss in Section 4.3, some speedups could be observed in particular benchmarks.

Benchmarks. We tested our implementation on the 191 benchmarks from the LLVM Test Suite. The test suite includes benchmarks such as MiBench, Shootout and Polybench. We have augmented this suite with the integer benchmarks written in C from SPEC CPU2017 [Bucek et al. 2018]; thus, obtaining 16 large benchmarks such as gcc, blender and imagick. Table 1 shows statistics for these SPEC CPU2017 benchmarks. Some benchmarks have considerably large code (as gcc, with 10,405 functions). We have also counted the number of static contexts in these programs using a simple dynamic programming algorithm. 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 15,000 contexts. In Section 4.4 we analyze the impact of different cutoffs in our

optimization. Table 1 shows how many 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. 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 programs from SPEC CPU2006 when comparing to `clang -O2`.

Indirect Calls. For simplicity, we opted to disregard targets of indirect calls. Therefore, our implementation does not optimize dynamically invoked functions. In SPEC CPU2017, this specially affects `xlancbnk` and `omnetpp`. Because dynamic dispatch is used very early in the execution of these benchmarks, their CCT could not be accurately constructed in compile-time. For instance, `omnetpp` has 5,245 functions, but gave us only 5 contexts. 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. 1995b; Grove et al. 1997; Milanova et al. 2004; Shivers 1988].

4.1 RQ1 – Compilation Time

Table 2 shows how our technique affects the compilation time of different SPEC CPU2017 programs. Compilation time tends to increase. However, once we limit the number of contexts to 15,000, this growth does not cause compilation to be impractical. Considering all SPEC CPU2017 benchmarks (except `blender`, on which Full Cloning runs out of memory), vanilla `clang -O2` takes 15 minutes running link-time optimizations. `clang -Os` takes only 7:33 minutes. `clang -O2` running with our inline DFA technique (limited at 15,000 contexts) and optimizations take 3.13x longer than `-O2` alone. Full Cloning raises compilation times by 3.40x.

On `blender`, our approach incidentally yields shorter compilation time than `clang -O2` without it. We observed that, in this case, our DFA contained functions that were aggressively inlined by `clang -O2` during link-time, thus considerably growing the number of functions that were further optimized. In light of these results, we can answer our first research question positively: our DFA-based method yields practical compilation times, even for large programs.

4.2 RQ2 – Code Size

Figure 9 (Left) compares the size of binaries produced by `clang -O2` when augmented with our approach, and with full cloning. Out of a universe of 209 programs, our implementation of context-sensitive constant propagation finds optimization opportunities in 128. Finding such opportunities is not trivial, because `clang -O2` already runs a very extensive intra-procedural implementation of constant propagation. DFA-based code generation produces smaller binaries in 31 out of these 128 benchmarks. Cloning generates smaller binaries in 97 benchmarks. However, in most cases in which cloning is at advantage, the difference is very small (less than 10KB). In contrast, our method avoids very significant size growth when dealing with large benchmarks. For instance, in the `ClamV` benchmark, the output binary generated by the DFA-based method is 76MB smaller than the same binary optimized with traditional cloning. In total, the binaries produced by full cloning are 2.63 times larger than the binaries that we generate with our state machines. If we only consider the bytes *added* to the original binary (i.e. optimized binary size minus binary size generated by `clang -O2`), the difference is even larger: while our method adds 33.37MB in total to

Table 2. Binary sizes (B.) and compilation times (T.) on SPEC CPU2017 for LLVM -O2, LLVM -Os, the Inline DFA and Full Cloning (FC). Here, compilation time only counts link-time optimizations, when we run our optimization (compilation times before this are unaffected). On 526.blender, the largest benchmark in the suite, Full Cloning crashes, running out of memory (OOM) when generating code for the optimized program.

Benchmark	B. (O2)	T. (O2)	B. (Os)	T. (Os)	B. (DFA)	T. (DFA)	B. (FC)	T. (FC)
500.perlbench	2.76M	01:10	1.85M	00:40	4.99M	47:22	2.67M	62:24
502.gcc	12.44M	08:55	7.74M	03:32	12.48M	09:45	17.40M	19:59
505.mcf	29.58K	00:00	22.73K	00:00	497.86K	01:59	71.22M	112:14
508.namd	735.70K	00:18	514.52K	00:12	735.70K	00:18	735.53K	00:19
510.parest	2.24M	00:58	1.61M	00:40	2.24M	00:58	2.23M	00:59
511.povray	1.48M	00:36	856.97K	00:22	1.48M	00:36	1.48M	00:37
519.lbm	18.58K	00:00	18.58K	00:00	22.70K	00:00	22.50K	00:00
520.omnetpp	2.25M	00:31	1.42M	00:23	1.79M	00:30	1.78M	00:31
523.xalanbmk	5.62M	01:28	3.16M	00:58	4.41M	01:29	4.31M	01:33
526.blender	17.24M	16:21	12.81M	03:07	21.50M	07:02	OOM	OOM
531.deepsjeng	94.95K	00:02	63.66K	00:01	3.85M	01:24	6.21M	04:19
538.imagick	1.63M	00:54	1.27M	00:35	4.57M	07:35	3.58M	08:02
541.leela	110.75K	00:01	76.44K	00:01	96.91K	00:01	96.74K	00:02
544.nab	108.95K	00:02	84.38K	00:01	612.76K	01:02	74.64M	32:40
557.xz	158.19K	00:03	116.97K	00:02	170.22K	00:03	153.66K	00:03
997.specrand	10.32K	00:00	10.32K	00:00	10.32K	00:00	14.24K	00:00
999.specrand	12.31K	00:00	10.32K	00:00	10.32K	00:00	14.24K	00:00

all executables, cloning adds 281.94MB in order to implement the same optimization, i.e. it adds 8.5x more bytes.

Cloning also generates binaries that are smaller than our executables in a few cases. For instance, Figure 2 shows that in SPEC’s imagick, the binary generated by cloning is 22% smaller. We have manually inspected several of such cases, and found a common pattern. Our naïve optimization often specializes contexts that are provably unreachable. In the case of cloning, if the compiler can prove that a function is unreachable, that function will be eliminated, together with all the calling path that it originates. This shortcoming is not a fundamental limitation of our approach: it is still possible to integrate it with dead-code elimination—something that we have not done. Nevertheless, our technique is still substantially better than cloning in terms of code size. In Table 2, we observe clang -Os generating binaries that have a (geometric) mean size of 0.71x the same binaries generated by clang -O2. Using our DFA-based optimization on top of clang -O2, limiting it at 15,000 contexts, increases binary sizes by 1.79x (geo-mean). Again, Full Cloning has a significantly larger overhead, increasing binary size by 5.78x.

We find that compilation times and binary sizes are tightly linked. Both the DFA-based approach and Full Cloning take similar times to run their core algorithms, which all run on linear time on the size of the optimization tree. However, because Full Cloning introduces more functions, all other stages of optimization and code generation are heavily impacted, yielding the results we observe. With these observations, we can answer our research question positively: the DFA-based implementation of context-sensitive optimizations generates significantly smaller binaries than Full Cloning.

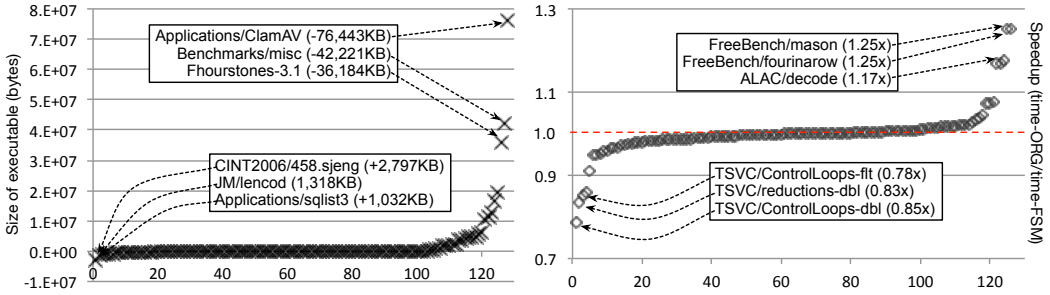


Fig. 9. (Left) Size comparison between our approach and full cloning. Each point is a benchmark. Y-axis shows the number of bytes in the executable generated by full cloning minus the same binary generated with our method. The higher the point, the better it is to our approach. (Right) The impact of our context-sensitive constant propagation on program speed. Each point is a benchmark. Y-axis shows the ratio between the running time of `clang -O2` and our approach (thus, 1.25x means 25% speed-up, and 0.85x means 15% slowdown). The greater the ratio, the better to our approach.

4.3 RQ3 – Running Time of Compiled Code

The goal of this section is to measure the overhead that our state machines add onto highly optimized executable programs. To this effect, Figure 9 (Left) compares the runtime of the executables produced by `clang -O2` plus either our state machines, or full cloning. Numbers are the arithmetic mean of five executions. Binaries produced with full cloning show no statistical difference to the binaries produced with the state machines. In total, programs optimized with the DFA-based method took 1517.4s seconds to run, versus 1515.4s seconds taken by the programs transformed with full cloning. These numbers are the average of five runs. Full cloning is less than 0.13% faster, and this difference yields a p-value above 0.05; hence, outside a confidence interval of 95%. In spite of the larger discrepancies that we can find when looking at individual benchmarks, the experiment lets us conclude that the runtime overhead of our state-machines and the dynamic dispatch of specialized code is minimum, when compared to more traditional approaches.

The Impact of State Machines onto Code Produced by `clang -O2`. Figure 9 (Left) compares the runtime of code that uses our context-sensitive version of constant propagation, and code that does not. Both programs have been optimized with `clang -O2`. Our implementation of constant propagation is not able to deliver a net gain on top of `clang -O2`. This result is not surprising: at this level, LLVM applies 228 analyses and optimizations onto programs. Some of these optimizations, namely inlining, already bring forward most of the benefits that constant propagation could provide. Furthermore, inlining enables very aggressive dead-code elimination, something that our version of context-sensitive constant propagation does not.

Nevertheless, Figure 9 (Left) shows that our state machines do not add any substantial overhead onto `clang -O2`. As with most optimizations, we have observed both speed-ups and slowdowns. We observed 11 consistent speed-ups of at least 3% with context-sensitive constant propagation (confidence of 95%), and 13 consistent slowdowns of at least 3%. Impact in running time ranged from 25% (speed-up) to 15% (slowdown). All 5 slowdowns greater than 10% come from the same benchmark suite: TSVC, which consists of the same program run with different flags. We have inspected this benchmark, and found that “there is a dummy function called in each loop to make all computations appear required” (quoted from a comment in the source code). When using the DFA-based method, calls to the dummy function, present in every loop, modify the DFA state. If we apply our optimization without inlining the code of the state machine, e.g., the transition

function, in the dummy function, then LLVM’s dead code elimination is also able to remove that function. In this case, we obtain the same positive effects as in full cloning.

In spite of the previous discussion, we have found speedup opportunities with our naïve context-sensitive constant propagation algorithm. For instance, in the Dhrystone benchmark we could produce a binary 32x faster than `clang -O2`. In this case, the combination of constant propagation, loop unrolling and dead code elimination let the compiler solve much of the computations in the benchmark statically. We emphasize that the goal of this specific experiment is not to speedup programs. Rather, we want to demonstrate that our state machines add a small overhead onto programs. 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. And, as Table 1 shows in the **Contexts** column, traditional implementations of clone-based optimizations would be too expensive to be applied in SPEC programs, for instance.

4.4 RQ4 – Scalability

Figure 10 shows how the size of programs vary once we change the number of contexts considered when performing context-sensitive constant propagation. The figure contrasts the DFA-based approach introduced in this paper with full cloning. Each line in Figure 10 represents a single benchmark. Points along the X-axis are the number of contexts considered. For this experiment, we have evaluated 5K, 10K, 15K, 20K and 25K contexts. Notice that although the number of contexts bear important impact on the size of binaries, it does not contribute noticeably to their running time. We have not been able to measure statistically significant variations in running time within a 90% confidence interval.

Linear growth. Figure 10 lets us draw two conclusions about the implementation of context-sensitive optimizations via state machines and via full cloning. First, in both cases, the size of the binary produced grows linearly with the number of contexts considered. This conclusion comes from a mixed-effects linear regression, where the groups correspond to benchmarks, the random effect is the benchmark-specific “size per context”, and the fixed effect is the number of contexts optimized. The coefficient of determination (R^2), in this case, is 0.97; hence, very close to a perfect linear behavior.

Lower linear coefficient. Second, Figure 10 makes it clear that the DFA-based approach presents much better scalability. Thus, although in both approaches the sizes of binaries grow linearly with the number of contexts considered, full cloning uses a much higher linear coefficient. This fact is evident from a visual inspection of line slopes presented in both the figures. In some cases, like in `gcc`, by going from 5K to 25K contexts, our implementation increases code size by a factor of only 2.72x (794KB to 2.17MB). However, full cloning leads to an increase of 57x (652KB to 37.80MB). This size explosion is due to the need to provide multiple versions not only of the optimized functions, but to whole paths in the call graph of the target program. Notice that both techniques: state machines and full cloning, track exactly the same paths in the program’s call graph. However, in the case of our technique, paths are recorded via a data-structure, whereas full cloning uses actual code to record these paths.

4.5 Discussion

This section has contrasted two approaches to implement context-sensitive optimizations. The first, the idea that we introduce in this paper, uses a combination of state-machine plus dynamic dispatching of optimized functions. The second approach clones paths within the call graph of the target program. In terms of code size, this approach is equivalent to unrestricted inlining of optimizable functions. The experiments in this section show that our state machines do not

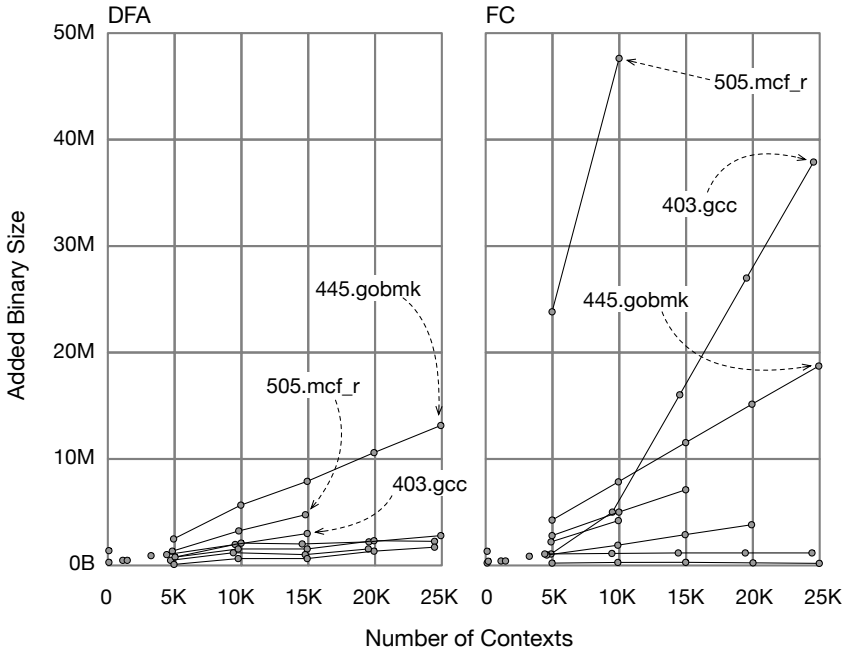


Fig. 10. Number of bytes added to the program’s binary when we vary the number of contexts in the optimization tree, on the SPEC benchmarks. Each benchmark is a line, with points resulting from limiting the number of contexts in the optimization tree to 5K, 10K, 15K, 20K and 25K. When constant propagation cannot find more optimization opportunities even with a larger limit of contexts considered, results collapse into a single point. On the left, we have the result of applying optimizations with our DFA-based approach, whereas results for Full Cloning (FC) are shown on the right.

impose extra runtime overhead onto the optimized programs. And, in contrast to the cloning-based alternative, they lead to binaries that are orders of magnitude smaller.

In small programs, this difference in code size tends to be irrelevant. There are even situations in which full-cloning allied with extensive-dead code elimination leads to smaller binaries. However, once we consider large programs, the gap between the two approaches becomes considerable. To give the reader some perspective on these numbers, we have applied both approaches onto the compilation of Mozilla’s Firefox version 80.0.1⁶. The whole project has more than 100K functions; however, most of them are in shared libraries, dynamically linked to the executables. The `firefox` executable yields only 922 functions visible at the LLVM LTO level. We find 456 optimizable contexts in this environment, pruning our optimization tree at 14,949 nodes and 15,144 DFA states. Adding this state machine to the executable increases its size from 7.2MB to 9.2MB. In contrast, the final binary produced by cloning has 34MB.

5 RELATED WORK

This work concerns the implementation of context-sensitive optimizations. Even though this is a well-known topic, being even discussed in general compiler textbooks, we believe that our contribution is unique, when compared against previous art. To the best of our knowledge, there are no

⁶Build instructions are available at https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Compiling_Firefox_With_Clang_On_Linux

implementation 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 often use context-sensitive information towards program improvement.

5.1 Context-Sensitive Analyses and Optimizations

The programming languages community has spent much effort into making context-sensitive analyses practical, as we have mentioned in Section 1. Thomas Reps has presented some of the first, and most important, results in this regard [Reps 1997, 2000]. In particular, Reps has shown that it is possible to improve interprocedural dataflow analyses by ruling out impossible paths between calls and returns. Paths that do not encode strings in a context-free language (CFL) should not be taken into consideration. These results refer to *static analyses*. Our work, in contrast, is related to the implementation of context-sensitive *optimizations*—which are enabled by those analyses. To this end, we create a state machine that tracks contexts dynamically. Said machine recognizes a subset of the context-free languages that Reps introduces. Thus, the undecidability results of Reps [1997] do not apply to the discussions in this paper.

Complete Context-Sensitive Analyses. State-of-the-practice techniques are able, today, to implement fully context-sensitive static program analyses. Watersheds, in this case, seem to have been the work of Whaley and Lam [2004], and the work of Zhu and Calman [2004]. They have shown, independently, that binary decision diagrams (BDD) were able to represent context-sensitive call-graphs with up to 10^{14} different paths. Key to that scalability was the ability of BDDs to merge information common among different call strings. Notice, however, that this kind of approach treats call strings context-sensitively, but does not effectively clone the heap. Nevertheless, context-sensitive information can still be used, for instance, whenever it is unique at a given program point, regardless of the call path reaching that point. For a comprehensive overview on the use of BDDs to track down calling contexts, we recommend the work of Lhoták and Hendren [2006].

Optimizations discussed in Section 2. Figure 2, Page 5, provides examples of four different ways to carry out context-sensitive optimizations. As discussed in Section 2, context-sensitive optimizations have been employed to perform type specialization in generic functions, to disambiguate pointers statically and to inline the body of function calls. Nevertheless, all these optimizations are used in limited ways, due to the problem of code-size explosion. In other words, thus far, either the optimizations are applied onto sites where context-sensitive information is invariant, or they must be restricted to small programs, or they are restricted to a few levels of the calling context tree.

5.2 Program Specialization via Function Cloning

Function cloning is a well-known way to enable code specialization. Compiler textbooks such as Kennedy and Allen [2002, pp.594]’s and Grune et al. [2012, pp.325]’s describe 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 on how they are chosen. Under such a classification, the method described in this paper falls on the first category, as clones are built during compilation time. However, we emphasize that none of the static techniques that we are aware of is able to deal with specialization in a fully context-sensitive way as we do: they clone functions only if a given call site has some invariant property, regardless of the call path that reaches it. In the rest of this section we go over the main techniques.

Fully Static Function Specialization. Static code specialization is the oldest and most well-known technique, among the three categories that we have listed above. The most extensive discussion about clone-based optimizations that we are aware of can be found in Mary Hall’s PhD dissertation [Hall 1991, Cp.5]. At that time, function cloning had yet to find space in industrial-strength compilers. In the words of Kennedy and Allen [2002], “*The Convex Applications Compiler* [Metzger

and Stroud 1993] is the only commercial compiler we know that performs this optimization". Today, function cloning has more users. For instance, gcc can produce specialized clones of a function, if in some specific contexts it receives only constants as parameters. This is a restricted version of Metzger and Stroud's inter-procedural constant propagation. 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 functions are specialized to particular types. Other examples along similar direction include the work of Dragos and Odersky [2009], Sallenave and Ducournau [2012] and Petrashko et al. [2016].

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. Dynamic and hybrid code specialization techniques also abound in the literature. Fully dynamic approaches are typical among just-in-time compilers [Gal et al. 2009; Hackett and Guo 2012; Hölzle et al. 1991; Santos et al. 2013]. For instance, Santos et al. [2013] and Lima et al. [2020] have proposed to generate specialized routines based on the runtime value of the arguments passed to JavaScript functions. These values can be easily inspected, because code is being generated while the program executes. This approach leads to similar results as our constant propagation, although it is done on-the-fly.

Hybrid Function Specialization. Hybrid specialization combines static and dynamic information to customize program parts. For instance, Samadi et al. [2012] and Tian et al. [2011] generate programs containing distinct routines to handle different kinds of inputs. Another example of hybrid specialization based on cloning concerns the line of work known as *Runtime Pointer Disambiguation* [Alves et al. 2015; Rus et al. 2002; Sperle Campos et al. 2016]. This technique consists in producing runtime 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 arguments cannot alias.

5.3 Context Sensitive Optimizations in JIT-Compilers

Just-in-time compilers are able to carry out context-sensitive optimizations, as a consequence of compiling code while said code runs. A visible effect of these optimizations are specializations, which can be based on single values [Lima et al. 2020; Santos et al. 2013], ranges of values [Sol et al. 2011] or types of values [Gal et al. 2009; Inoue et al. 2011]. Some JIT compilers limit the number of versions of each specialized function, like two in the case of Santos et al. [2013] or four as described by Lima et al. [2020]. Other compilers, notably trace-based, might keep a copy of binary code per calling context. In the words of Inoue et al. [2011], "*the same method called from many different places may appear in many traces because it achieves an effect equivalent to method inlining*". The work described in this paper has been proposed, implemented and evaluated on a static compiler. Nevertheless, we believe that the combination of state machines and dynamic dispatch can also be used to mitigate code size explosion in JIT-compilers, specially on method-based systems that resort to extensive inlining, like the one used in the IBM J9 VM [Grcevski et al. 2004].

5.4 Context Numbering

Previous work have discussed the possibility of assigning a different identifier, e.g., a number, to every context that could exist during the execution of a program. Such identifiers support, for instance, more precise program profiling and event logging. The state-of-the-art approach in this regard is the work of Sumner et al. [2010, 2012]. Their numbering approach is inspired by Ball and Larus [1996]'s path numbering algorithm. There is a beautiful correlation between Sumner et al.'s work and ours: we could use their technique to select the right version of an optimized function to dispatch. To carry out this approach, two actions are in order: (i) instrument the program to

update the context ID at every function call; and (ii) use a table to associate context IDs with functions. To work correctly in our setting, Sumner et al.'s original implementation would require a small adaptation, for it is restricted to calling context trees (CCT) that can be numbered with 32-bit identifiers. Nevertheless, this is not the approach that we use in this paper to choose the right function to dispatch. Instead of assigning unique IDs to paths in the program's CCT, we assign unique IDs to the children of nodes in that tree. Because the number of children of CCT nodes is small, our universe of IDs can be much smaller than Sumner et al.'s. Moreover, instead of relying on a table, we use an automaton to keep track of the current path in the CCT.

In addition to Sumner et al.'s technique, which is exact, the programming languages literature contains probabilistic techniques to number calling contexts [Bond et al. 2010; Bond and McKinley 2007; Huang and Bond 2013]. Probabilistic numbering has been shown to be useful, for instance, to support sampling profilers. Given the probabilistic nature of the technique, there exists some chance that different contexts be assigned the same number. Therefore, in contrast to Sumner et al.'s or our approach, probabilistic numbering, if used to enable code specialization, would require some way to detect and resolve collisions. All this said, we emphasize that Sumner et al.'s exact technique or the probabilistic approaches have never been used as a means to implement context sensitive optimizations—the goal of this paper. Rather, they propose ways to assign unique IDs to calling contexts. In contrast, our key contribution is not a methodology to identify contexts at runtime, but rather the insight of combining such strategy with the dynamic dispatch of specialized code.

6 CONCLUSION

In this paper, we presented a method for implementing context-sensitive optimizations using dynamic dispatch. Traditionally, context tracking had been accomplished by copying code, either through function cloning or inlining. By tracking contexts at running time, with the aid of a state machine, we were able to generate less code in order to implement the same optimizations. Our method is optimization-agnostic, since any context-sensitive optimization can be described by an Optimization Tree—a data structure we introduce. This means that our work complements recent efforts from the community to create scalable context-sensitive analyses and optimizations. We implemented our ideas in LLVM, combining them with a simple context-sensitive constant propagation. Experimental results showed that the overhead introduced by the state machine that tracks calling contexts is negligible, since the performance of the code generated by our method was in practice equivalent to that obtained with traditional function cloning. However, we were able to generate smaller binaries than what traditional function specialization produces, due to the ability to avoid cloning functions which were not optimized. Finally, the fact that we were able to observe speed-ups in large real-world programs when comparing to `clang -O2` suggests that context-sensitive optimizations remain largely unexplored by industrial compilers.

Software. Our implementation is publicly available at <https://github.com/gpoesia/eos>.

ACKNOWLEDGMENTS

We thank Breno Guimarães for helping with the implementation of inter-procedural constant propagation early on in this project, and the anonymous reviewers for their helpful feedback, which greatly improved our evaluation. Fernando Pereira is sponsored by CNPq (Grant 406377/2018-9) and FAPEMIG (Grant PPM-00193-16). This project was developed while Gabriel Poesia was a master student in the Graduate Computer Science Program of UFMG. During that time, he was sponsored by a scholarship from Google Research in Latin America.

REFERENCES

- Pérciles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. In *OOPSLA*. ACM, New York, NY, USA, 589–606.
- Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oteau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*. ACM, New York, NY, USA, 259–269.
- Giorgio Ausiello, Camil Demetrescu, Irene Finocchi, and Donatella Firmani. 2012. k-Calling Context Profiling. In *OOPSLA*. ACM, New York, NY, USA, 867–878.
- Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *MICRO*. IEEE Computer Society, USA, 46–57.
- Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. 2010. Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses. In *PLDI*. ACM, New York, NY, USA, 13–24.
- Michael D. Bond and Kathryn S. McKinley. 2007. Probabilistic Calling Context. In *OOPSLA*. ACM, New York, NY, USA, 97–112.
- Matt Brown and Jens Palsberg. 2017. Jones-optimal Partial Evaluation by Specialization-safe Normalization. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017), 14:1–14:28.
- James Bucek, Klaus-Dieter Lange, and JÓakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *ICPE*. Association for Computing Machinery, New York, NY, USA, 41–42. <https://doi.org/10.1145/3185768.3185771>
- Keith D Cooper, Mary W Hall, and Ken Kennedy. 1993. A Methodology for Procedure Cloning. *Comput. Lang.* 19, 2 (1993), 105–117.
- Dibyendu Das. 2003. Function inlining versus function cloning. *ACM SIGPLAN Notices* 38, 6 (2003), 23–29.
- Jeffrey Dean, Craig Chambers, and David Grove. 1995a. Selective Specialization for Object-Oriented Languages. In *PLDI*. ACM, New York, NY, USA, 93–102. <https://doi.org/10.1145/207110.207119>
- Jeffrey Dean, David Grove, and Craig Chambers. 1995b. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*. Springer-Verlag, London, UK, UK, 77–101.
- Iulian Dragos and Martin Odersky. 2009. Compiling Generics Through User-directed Type Specialization. In *ICOOOLPS*. ACM, New York, NY, USA, 42–47.
- Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *PLDI*. ACM, New York, NY, USA, 242–256.
- Manuel Fähndrich, Jakob Rehof, and Manuvir Das. 2000. Scalable Context-sensitive Flow Analysis Using Instantiation Constraints. In *PLDI*. ACM, New York, NY, USA, 253–263.
- Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *FSE*. ACM, New York, NY, USA, 576–587.
- Olivier Fluckiger, Guido Chari, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 1 (2020), 36 pages.
- Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *PLDI*. ACM, New York, NY, USA, 465–478.
- Rakesh Ghiya and Laurie J. Hendren. 1996. Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C. In *POPL*. ACM, New York, NY, USA, 1–15.
- Nikola Grcovski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundareshan. 2004. Java™ Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *VM*. USENIX Association, USA, 12.
- David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-Oriented Languages. In *OOPSLA*. ACM, New York, NY, USA, 108–124. <https://doi.org/10.1145/263698.264352>
- Dick Grune, Kees van Reeuwijk, Henri E. Baland Cerial J. H. Jacobs, and Koen Langendoen. 2012. *Modern Compiler Design* (2nd ed.). Springer, London, UK, UK.
- Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *PLDI*. ACM, New York, NY, USA, 239–250.
- Mary Wolcott Hall. 1991. *Managing interprocedural optimization*. Ph.D. Dissertation. Rice University, Houston, TX, USA. UMI Order No. GAX91-36029.
- Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.* 21, 4 (1999), 848–894.
- Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP*. Springer-Verlag, London, UK, UK, 21–38.
- Jipeng Huang and Michael D. Bond. 2013. Efficient Context Sensitivity for Dynamic Analyses via Calling Context Uptrees and Customized Memory Management. In *OOPSLA*. ACM, New York, NY, USA, 53–72.

- Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2011. A Trace-Based Java JIT Compiler Retrofitted from a Method-Based Compiler. In *CGO*. IEEE Computer Society, USA, 246–256.
- Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (2017), 28 pages. <https://doi.org/10.1145/3133924>
- Ken Kennedy and John R. Allen. 2002. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. 2009. *Data Flow Analysis: Theory and Practice* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.
- Anton Korobeynikov. 2007. Improving Switch Lowering for the LLVM Compiler System. In *SYRCoSE*. RAS, Innopolis, Russia, A.I-A.V.
- Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE, Washington DC, 75–88.
- Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *PLDI*. ACM, New York, NY, USA, 278–289.
- David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *CGO*. ACM, New York, NY, USA, 126–137.
- Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *CC*. Springer, Berlin, Heidelberg, 47–64.
- Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and Scalable Context-sensitive Pointer Analysis via Value Flow Graph. In *ISMM*. ACM, New York, NY, USA, 85–96.
- Yue Li, Tian Tan, Anders Moller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *TOPLAS To-Appear*, 1 (2020), 1–40.
- Caio Lima, Junio Cezar R. da Silva, Guilherme V. Leobas, Erven Rohou, and Fernando Magno Quintão Pereira. 2020. Guided just-in-time specialization. *Sci. Comput. Program.* 185, Article 2 (2020), 39 pages.
- Robert Metzger and Sean Stroud. 1993. Interprocedural constant propagation: an empirical study. *ACM Lett. Program. Lang. Syst.* 2, 1-4 (1993), 213–232.
- Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis. In *PLDI*. ACM, New York, NY, USA, 305–315.
- Ana Milanova. 2007. Light Context-sensitive Points-to Analysis for Java. In *PASTE*. ACM, New York, NY, USA, 25–30.
- Ana Milanova, Wei Huang, and Yao Dong. 2014. CFL-reachability and Context-sensitive Integrity Types. In *PPPJ*. ACM, New York, NY, USA, 99–109.
- Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2004. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engg.* 11, 1 (2004), 7–26.
- Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *PLDI*. ACM, New York, NY, USA, 475–484.
- Dmitry Petrashko, Vlad Ureche, Ondřej Lhoták, and Martin Odersky. 2016. Call Graphs for Languages with Parametric Polymorphism. In *OOPSLA*. ACM, New York, NY, USA, 394–409.
- Gabriel Poesia. 2017. *Dispatch of Context-Sensitive Optimizations*. Master’s thesis. Federal University of Minas Gerais.
- Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static Placement of Computation on Heterogeneous Devices. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 50:1–50:28.
- Thomas Reps. 1997. Program Analysis via Graph Reachability. In *ILPS*. MIT Press, Cambridge, MA, USA, 5–19.
- Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. *TOPLAS* 22, 1 (Jan. 2000), 162–186.
- Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2002. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. In *ICS*. ACM, New York, NY, USA, 274–284.
- Olivier Sallénave and Roland Ducournau. 2012. Lightweight Generics in Embedded Systems Through Static Analysis. In *LCTES*. ACM, New York, NY, USA, 11–20.
- Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. 2012. Adaptive Input-aware Compilation for Graphics Engines. In *PLDI*. ACM, New York, NY, USA, 13–22.
- Henrique Nazare Santos, Pericles Alves, Igor Costa, and Fernando Magno Quintão Pereira. 2013. Just-in-time Value Specialization. In *CGO*. IEEE Computer Society, Washington, DC, USA, 1–11. <https://doi.org/10.1109/CGO.2013.6495006>
- O. Shivers. 1988. Control Flow Analysis in Scheme. In *PLDI*. ACM, New York, NY, USA, 164–174.
- Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza Andrade da Silva Bigonha. 2011. Dynamic Elimination of Overflow Tests in a Trace Compiler. In *CC*. Springer-Verlag, London, UK, UK, 2–21. https://doi.org/10.1007/978-3-642-19861-8_2

- Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL (2019), 48:1–48:29. <https://doi.org/10.1145/3290361>
- Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *ECOOP*. Springer, London, UK, UK, 22:1–22:26. <https://doi.org/10.4230/LIPIcs.ECOOP.2016.22>
- Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restriction of Function Arguments. In *CC*. ACM, New York, NY, USA, 163–173.
- Manu Sridharan and Rastislav Bodik. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *PLDI*. ACM, New York, NY, USA, 387–400.
- Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. In *DLS*. ACM, New York, NY, USA, 84–95. <https://doi.org/10.1145/2989225.2989236>
- Nicolas Stucki and Vlad Ureche. 2013. Bridging Islands of Specialized Code Using Macros and Reified Types. In *SCALA*. ACM, New York, NY, USA, 10:1–10:4.
- William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. 2010. Precise Calling Context Encoding. In *ICSE*. ACM, New York, NY, USA, 525–534. <https://doi.org/10.1145/1806799.1806875>
- William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. 2012. Precise Calling Context Encoding. *IEEE Trans. Software Eng.* 38, 5 (2012), 1160–1177. <https://doi.org/10.1109/TSE.2011.70>
- Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-Contexts Based Whole-Program Heap Analyses. In *Compiler Construction*. ACM, New York, NY, USA, 135–146.
- Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Compiler Construction*. ACM, New York, NY, USA, 27–38. <https://doi.org/10.1145/3377555.3377902>
- Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. *SIGPLAN Not.* 52, 6 (2017), 263–277. <https://doi.org/10.1145/3140587.3062359>
- Kai Tian, Eddy Zhang, and Xipeng Shen. 2011. A Step Towards Transparent Integration of Input-consciousness into Dynamic Program Optimizations. In *OOPSLA*. ACM, New York, NY, USA, 445–462.
- Vlad Vergu and Eelco Visser. 2018. Specializing a Meta-Interpreter: JIT Compilation of Dynsem Specifications on the Graal VM. In *ManLang*. ACM, New York, NY, USA, 1–14. <https://doi.org/10.1145/3237009.3237018>
- Haichuan Wang, Peng Wu, and David Padua. 2014. Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization. In *CGO*. ACM, New York, NY, USA, 295:295–295:305.
- Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *ECOOP*. Springer, London, UK, UK, 712–734.
- John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*. ACM, New York, NY, USA, 131–144.
- Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *PLDI*. ACM, New York, NY, USA, 1–12.
- Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by Level: Making Flow- and Context-Sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *CGO*. ACM, New York, NY, USA, 218–229. <https://doi.org/10.1145/1772954.1772985>
- Jianwen Zhu and Silvian Calman. 2004. Symbolic Pointer Analysis Revisited. In *PLDI*. ACM, New York, NY, USA, 145–157.