On-Stack Replacement, Distilled

Daniele Cono D'Elia
delia@dis.uniroma1.it
Sapienza University of Rome
Italy

Camil Demetrescu
demetres@dis.uniroma1.it
Sapienza University of Rome
Italy

Abstract

On-stack replacement (OSR) is essential technology for adaptive optimization, allowing changes to code actively executing in a managed runtime. The engineering aspects of OSR are well-known among VM architects, with several implementations available to date. However, OSR is yet to be explored as a general means to transfer execution between related program versions, which can pave the road to unprecedented applications that stretch beyond VMs. We aim at filling this gap with a constructive and provably correct OSR framework, allowing a class of general-purpose transformation functions to yield a special-purpose replacement. We describe and evaluate an implementation of our technique in LLVM. As a novel application of OSR, we present a feasibility study on debugging of optimized code, showing how our techniques can be used to fix variables holding incorrect values at breakpoints due to optimizations.

CCS Concepts • **Software and its engineering → Compilers**; *Correctness*; *Just-in-time compilers*; Software testing and debugging;

Keywords Dynamic compilers, deoptimization, debugging.

ACM Reference Format:
Daniele Cono D'Elia and Camil Demetrescu. 2018. On-Stack Replacement, Distilled. In *PLDI '18: ACM SIGPLAN Conference on Programming Language Design and Implementation, June 18–22, 2018, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192396

1 Introduction

On-stack replacement (OSR) is a mechanism employed by modern runtimes to support on-the-fly transitions between variants of a currently executing function. OSR was first prototyped in the SELF VM [24] and has since become a fundamental element of multi-tier architectures in advanced runtimes. OSR is commonly used to transfer the execution between a base version of a function, which either is being interpreted or a baseline compiler has produced code for it, to a more optimized version, usually generated by a dynamic compiler based on the run-time behavior. As speculative optimizations are essential for high performance, deoptimization may become necessary: it can be supported either by reconstructing a frame for the interpreter or the baseline compiled code, which both have a fixed stack layout, or by performing OSR into dynamically generated code as in Jikes RVM [15]. Following the classification proposed in [29], we use OSR to refer to both optimizing and deoptimizing transitions.

A multi-tier approach increases the implementation and maintenance costs for a VM. Compilers have to encode the metadata and glue code needed to get the program state to a correct resumption point. For instance, unoptimized frame reconstruction normally requires the creation of scope descriptors accounting for program location and live variables at each deoptimization point. Deoptimization entry points may also be placed manually in the base tier as in V8 [45]. Also, restrictions may be put on optimizations to preserve the feasibility of OSR at specific points.

As a key motivation for our work, we argue that generalizing OSR to support transitions at arbitrary program points in the face of common classes of code transformations may not only be of interest for a VM implementor, but more importantly pave the way to unprecedented interesting use cases for OSR. For instance, one wishes to collect information about a program crash in an optimized production environment: OSR could be used to revert the state so that a core dump would reflect the expected behavior at the source level, reporting correct values for variables that are live at the crash point. As a second example, a program can be obfuscated to prevent security attacks via dynamic diversity by randomly diverting execution between different program versions at arbitrary execution points. Furthermore, OSR could be employed to switch between instrumented and uninstrumented code in tools that perform IR instrumentation such as dynamic memory safety checkers.

Contributions and Overview. In this paper we contribute to the theory and practice of OSR with a framework that explores soundness and flexibility of OSR transitions between different versions of a program in the face of common code transformations. The techniques presented here can be implemented in a general-purpose compiler with a modest

PLDI '18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192396

effort, and may provide useful building blocks to compiler architects. Our contributions are three-fold:

- We distill the essence of OSR to an abstract program morphing problem: code transformations can be made OSR-aware in isolation, and then composed in arbitrary ways. For a class of common optimizations, we devise an algorithm to generate the state realignment code possibly required for a transition to take place. A key observation is that for several classes of code transformations it is enough to reconstruct just the values of live variables rather than the entire state.
- We show that light changes are needed to accommodate our techniques in a general-purpose compiler. We discuss an implementation of our ideas in LLVM, and assess it over code drawn from popular benchmarks.
- We present a feasibility study of a novel OSR application in optimized-code debugging. While optimizations can cause variables to hold inconsistent values at breakpoints, our techniques are able to reconstruct most values expected at the source level.

Section 2 introduces formal preliminaries. Section 3 and 4 describe our OSR theoretical framework. An implementation is discussed in Section 5 and evaluated in Section 6. The feasibility study is presented in Section 7. Section 8 addresses related work. Section 9 presents directions for future work.

2 Language Framework

In this section we introduce some basic definitions used in our representation of programs and code transformations.

2.1 Program Syntax and Semantics

Our discussion is based on a minimal imperative language whose syntax is reported in Figure 1.

Definition 2.1 (Program)**.** A program is a sequence of instructions of the form:

$$\mathrm{P} = \langle I_1, I_2, \ldots, I_n \rangle \in \mathit{Prog} = \bigcup_{i=2}^{\infty} \mathit{Instr}^i$$

where:

- $I_i \in \mathit{Instr}$ is the i-th instruction of the program, indexed by program point $i \in [1, n]$
- $I_1 = \texttt{in} \cdots$ is the initial instruction
- $\forall i \in [2, n-1] : I_i \neq \texttt{in} \cdots \wedge I_i \neq \texttt{out} \cdots$
- $I_n = \texttt{out} \cdots$ is the final instruction

Instruction `in` must appear at the beginning of a program and specifies the variables that must be defined prior to entering the program. Similarly, `out` occurs at the end and specifies the variables that are returned as output. We indicate by `e[x]` that `x` is a variable of the expression $\mathtt{e} \in \mathit{Expr}$, and by $|\mathrm{P}| = n$ the number of instructions in $\mathrm{P} = \langle I_1, I_2, \ldots, I_n \rangle$.

In order to provide a formal semantics, we need to introduce definitions for program state and memory store:

$$\begin{array}{rcl}
\mathit{Instr} & ::= & \mathit{Var} \texttt{ := } \mathit{Expr} \\
& & \mid \texttt{if (} \mathit{Expr} \texttt{) goto } \mathit{Num} \\
& & \mid \texttt{goto } \mathit{Num} \\
& & \mid \texttt{skip} \mid \texttt{abort} \\
& & \mid \texttt{in } \mathit{Var} \cdots \mathit{Var} \mid \texttt{out } \mathit{Var} \cdots \mathit{Var} \\
\mathit{Expr} & ::= & \mathit{Num} \mid \mathit{Var} \mid \mathit{Expr} \texttt{ + } \mathit{Expr} \mid \ldots \\
\mathit{Var} & ::= & \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \mid \ldots \\
\mathit{Num} & ::= & \ldots \mid \texttt{-2} \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots
\end{array}$$

Figure 1. Program Syntax

$$\frac{I_l = \mathtt{x{:=}e} \wedge (\sigma, \mathtt{e}) \Downarrow v}{(\sigma, l) \Rightarrow_{\mathrm{P}} (\sigma[\mathtt{x} \leftarrow v], l+1)} \quad (1)$$

$$\frac{I_l = \texttt{goto m}}{(\sigma, l) \Rightarrow_{\mathrm{P}} (\sigma, \mathtt{m})} \quad (2)$$

$$\frac{I_l = \texttt{skip}}{(\sigma, l) \Rightarrow_{\mathrm{P}} (\sigma, l+1)} \quad (3)$$

$$\frac{I_l = \texttt{if (e) goto m} \wedge (\sigma, \mathtt{e}) \Downarrow 0}{(\sigma, l) \Rightarrow_{\mathrm{P}} (\sigma, l+1)} \quad (4)$$

$$\frac{I_l = \texttt{if (e) goto m} \wedge (\sigma, \mathtt{e}) \Downarrow v \wedge v \neq 0}{(\sigma, l) \Rightarrow_{\mathrm{P}} (\sigma, \mathtt{m})} \quad (5)$$

$$\frac{I_1 = \texttt{in x y} \cdots \wedge \sigma(\mathtt{x}) \neq \bot \wedge \sigma(\mathtt{y}) \neq \bot \wedge \cdots}{(\sigma, 1) \Rightarrow_{\mathrm{P}} (\sigma, 2)} \quad (6)$$

$$\frac{I_n = \texttt{out x y} \cdots \wedge \sigma(\mathtt{x}) \neq \bot \wedge \sigma(\mathtt{y}) \neq \bot \wedge \cdots}{(\sigma, n) \Rightarrow_{\mathrm{P}} (\sigma|_{\{\mathtt{x}, \mathtt{y}, \cdots\}}, n+1)} \quad (7)$$

Figure 2. Big-step semantics for the language of Figure 1. Transition relation $\Rightarrow_{\mathrm{P}} \subseteq \mathit{State} \times \mathit{State}$ is defined using meta-variables $\mathtt{x}, \mathtt{y} \in \mathit{Var}$, $\mathtt{e} \in \mathit{Expr}$, and $\mathtt{m} \in \mathit{Num}$. For a transition to apply, we assume that I_l is defined, i.e., $l \in [1, n]$.

Definition 2.2 (Memory Store)**.** A *memory store* is a total function $\sigma : \mathit{Var} \to \mathbb{Z} \cup \{\bot\}$ that associates integer values to defined variables, and $\bot$ to undefined variables. We denote by Σ the set of all possible memory stores.

By $\sigma[\mathtt{x} \leftarrow v]$ we denote the same memory store function as σ, except that `x` takes value v. Furthermore, for any $A \subseteq \mathit{Var}$, $\sigma|_A$ denotes σ restricted to the variables in A, i.e., $\sigma|_A(\mathtt{x}) = \sigma(\mathtt{x})$ if $\mathtt{x} \in A$ and $\sigma|_A(\mathtt{x}) = \bot$ if $\mathtt{x} \notin A$.

Definition 2.3 (Program State)**.** The *state* of a program $\mathrm{P} = \langle I_1, I_2, \ldots, I_n \rangle$ is described by a pair (σ, l), where σ is a memory store and $l \in [1, n]$ is the program point of the next instruction to be executed. We denote by $\mathit{State} = \Sigma \times \mathbb{N}$ the set of all possible program states.

In Figure 2 we provide a big-step semantics for the language using the transition relation $\Rightarrow_{\mathrm{P}} \subseteq \mathit{State} \times \mathit{State}$ that specifies how a single instruction of a program P affects its state. Our description relies on the relation $\Downarrow \subseteq (\Sigma \times \mathit{Expr}) \times \mathbb{Z}$ to describe how expressions are evaluated in a given memory store.

Definition 2.4 (Program Semantic Function)**.** We define the semantic function $[\![\mathrm{P}]\!] : \Sigma \to \Sigma$ of a program P as:

$$\forall \sigma \in \Sigma : \; [\![\mathrm{P}]\!](\sigma) = \sigma' \iff (\sigma, 1) \Rightarrow_{\mathrm{P}}^{*} (\sigma', |\mathrm{P}| + 1)$$

where $\Rightarrow_{\mathrm{P}}^{*}$ is the transitive closure of $\Rightarrow_{\mathrm{P}}$.

Note that a program has undefined semantics if its execution on a given store does not reach the final `out` instruction. This

accounts for infinite loops, abort instructions, exceptions, and ill-defined programs or input stores. We define the notion of program semantic equivalence as follows:

Definition 2.5 (Program Equivalence)**.** Two programs $\mathtt{p}_1$ and $\mathtt{p}_2$ are *semantically equivalent* iff $[\![\mathtt{p}_1]\!] = [\![\mathtt{p}_2]\!]$.

Finally, we provide a definition of trace of a transition system that will be useful to reason about feasible executions:

Definition 2.6 (Traces)**.** For a transition system $(S, R \subseteq S^2)$ a *trace* starting at $s \in S$ is a sequence $\tau = \langle s_0, s_1, \ldots, s_i, \ldots \rangle$ such that $s_0 = s$ and $\forall i \geq 0 : \; s_i \in \tau \; \wedge \; s_i \; R \; s_{i+1} \iff s_{i+1} \in \tau$. We denote by $\mathcal{T}_{R,s}$ the system of all traces starting from s.

For the deterministic transition relation $\Rightarrow_{\mathtt{p}}$ of our language, the system of traces $\mathcal{T}_{\Rightarrow_{\mathtt{p}},(\sigma,1)}$ of the execution transition system $(Store, \Rightarrow_{\mathtt{p}})$ contains a single trace starting at state $(\sigma, 1)$ for any initial store σ. We denote such trace by $\tau_{\mathtt{p}\sigma}$.

2.2 Program Properties and Transformations

In this section we present a formalism based on *computation tree logic* (CTL) that is amenable to automated reasoning [9]. The same formalism can be used to reason about program properties and to express program transformations.

Program Properties. A property can be expressed using a Boolean formula with free meta-variables that combine facts that must hold globally or at certain program points. A *model checker* can check formulas against concrete programs: for any program p and formula ϕ, the checker verifies whether there exists a substitution θ that binds free meta-variables with program objects so that $\theta(\phi)$ is satisfied in p. By $\mathcal{A} \models \phi$ we mean that structure $\mathcal{A}$ is a model of formula ϕ as in [9].

Analyses that involve finite maximal paths in the control flow graph (CFG) such as liveness and dominance can be expressed using first-order CTL operators to specify properties of CFG nodes and paths. In particular, temporal CTL operators can be used to express properties of some or all possible future computational paths, any one of which might be an actual path that is realized. We say that for any point l in a program p and two formulas ϕ and ψ, the following predicates are satisfied at l if ϕ holds:

- $\overrightarrow{AX}(\phi)$: for all immediate successors of l;
- $\overrightarrow{EX}(\phi)$: for at least one immediate successor of l;
- $\overrightarrow{A}(\phi \; U \; \psi)$: on all paths from l, until ψ holds;
- $\overrightarrow{E}(\phi \; U \; \psi)$: on at least one path from l, until ψ holds.

Corresponding operators $\overleftarrow{AX}$ and $\overleftarrow{EX}$ are defined for immediate predecessors of l, while $\overleftarrow{A}$ and $\overleftarrow{E}$ refer to backward paths from l. Operators A and E are quantifiers over paths, while X and U path-specific quantifiers. Notice that an expression of the form $\phi \; U \; \psi$ requires that ψ has to hold in the future.

Figure 3 shows a number of local predicates that will be useful throughout this paper. For instance, $\mathtt{p}, l \models \mathtt{lives(x)}$ holds iff on all backward paths ($\overleftarrow{A}$) starting at all the predecessors ($\overleftarrow{AX}$) of l, x has been defined somewhere, and there is at least one forward path from l that eventually reads x.

This allows us to define the set of live variables at any given point of a program, which will play a central role in this paper.

$$
\begin{array}{rcl}
\mathtt{def(x)} & \triangleq & I_l = \mathtt{x:=e} \;\vee\; I_l = \mathtt{in} \cdots \mathtt{x} \cdots \\
 & & [\mathtt{x} \textit{ is defined by instruction } I_l \textit{ in } \mathtt{p}] \\
\mathtt{use(x)} & \triangleq & I_l = \mathtt{y:=e[x]} \;\vee\; I_l = \mathtt{if\ (e[x])\ goto\ m} \;\vee \\
 & & I_l = \mathtt{out} \cdots \; [\mathtt{x} \textit{ is used by instruction } I_l \textit{ in } \mathtt{p}] \\
\mathtt{stmt}(I) & \triangleq & I = I_l \; [I \textit{ is the instruction at } l \textit{ in } \mathtt{p}] \\
\mathtt{point(m)} & \triangleq & \mathtt{m} = l \; [\textit{program point } \mathtt{m} \textit{ is } l \textit{ in } \mathtt{p}] \\
\mathtt{trans(e)} & \triangleq & I_l = \mathtt{x:=e'} \;\wedge\; \neg\mathtt{freevar(x,e)} \;\vee I_l \neq \mathtt{x:=e'} \\
 & & [\textit{no constituent of } \mathtt{e} \textit{ is modified by instr. } I_l \textit{ in } \mathtt{p}] \\
\mathtt{lives(x)} & \triangleq & \overleftarrow{AX}\overleftarrow{A}(\mathrm{true}\; U\; \mathtt{def(x)}) \wedge \overrightarrow{E}(\neg\mathtt{def(x)}\; U\; \mathtt{use(x)}) \\
 & & [\mathtt{x} \textit{ is live at program point } l \textit{ in } \mathtt{p}]
\end{array}
$$

Figure 3. Predicates expressing local properties of a point $l \in [1, n]$ in a program $\mathtt{p} = \langle I_1, \ldots, I_n \rangle$, with meta-variables $\mathtt{e}, \mathtt{e'} \in Expr$, $\mathtt{x}, \mathtt{y} \in Var$, and $l, \mathtt{m} \in Num$.

This allows us to define the set of live variables at any given point of a program, which will play a central role in this paper.

Definition 2.7 (Live Variables)**.** The set of live variables of a program p at point l is defined as:

$$\mathtt{live}(\mathtt{p}, l) \triangleq \{\, \mathtt{x} \in Var : \mathtt{p}, l \models \mathtt{lives(x)} \,\}$$

Later on we will also use two global predicates: `freevar(x, e)`, which holds iff `x` is a free variable of expression `e`, and `conlit(c)`, which holds iff expression `c` is a constant literal.

Program Transformations. To describe transformations, we use rewrite rules with side conditions drawn from CTL formulas in a similar manner to [26, 28]. We consider generalized rules that transform multiple instructions at once.

Definition 2.8 (Rewrite Rule)**.** A rule T has the form:

$$T = \quad m_1 : \hat{I}_1 \Longrightarrow \hat{I'}_1 \quad \cdots \quad m_r : \hat{I}_r \Longrightarrow \hat{I'}_r \quad \mathtt{if}\ \phi$$

where $\forall k \in [1, r]$, m_k is a meta-variable that denotes a program point, $\hat{I}_k$ and $\hat{I'}_k$ are program instruction patterns that can contain meta-variables, and ϕ is a side condition that states whether the rule can be applied to the input program.

The following example encodes a trivial peephole optimization based on a weak form of operator strength reduction [10], using three meta-variables `m`, `x`, and `y`:

$$m : \; \mathtt{y := 2 * x} \implies \mathtt{y := x + x}\ \mathtt{if}\ \mathit{true}$$

Rules can be applied to concrete programs by a transformation engine based on model checking. When a substitution θ can bind free meta-variables with program objects so that $\theta(\phi)$ is satisfied in p and $\theta(\hat{I}_k) = I_{\theta(m_k)} \in \mathtt{p}$ for some $k \in [1, r]$, then $I_{\theta(m_k)}$ is replaced with $\theta(\hat{I'}_k) = I'_{\theta(m_k)} \in \mathtt{p}'$:

Definition 2.9 (Rule Semantics)**.** Let T be a rewrite rule as in Definition 2.8. The transformation function $\lceil T \rceil : Prog \to Prog$ is defined as follows:

$$\forall \mathrm{P}, \mathrm{P}' \in \mathit{Prog} : \mathrm{P}' = \lceil T \rceil(\mathrm{P}) \iff \exists\, \theta :\ \mathrm{P} \models \theta(\phi) \wedge$$
$$\forall k \in [1, r] : \theta(\hat{I}_k) = I_{\theta(m_k)} \in \mathrm{P} \ \wedge\ \theta(\hat{I}'_k) = I'_{\theta(m_k)} \in \mathrm{P}'$$

We say that T is *semantics-preserving* if for any program P it holds $[\![\mathrm{P}]\!] = [\![\mathrm{P}']\!]$, where $\mathrm{P}' = \lceil T \rceil(\mathrm{P})$.

3 On-Stack Replacement Framework

OSR embodiments have to ensure that, whenever control is transferred from a currently running function version to another one, execution can transparently continue without altering the intended program semantics. In the adaptive optimization practice, optimizing OSR transitions typically happen at places where state realignment is simple, i.e., at a method or loop entry. The placement of deoptimizing OSR points is determined by the runtime: it can emit them for all instructions that might deoptimize [37], or group them and resume execution from the last instruction in the deoptimized code that causes outside-visible effects [49].

A framework for reasoning on the feasibility and soundness of state transfers could not only be valuable in generalized OSR applications such as those outlined in Section 1, but provide novel ideas to VM architects as well. Previous work has shown that performing OSR at more points can enable additional optimization opportunities [13]. Similar considerations can be made for deoptimization in multi-tier VMs: for instance, current strategies may hamper optimizations in the deoptimization target code, such as those that make new values alive across deoptimization points [45].

We propose a model based on *OSR mappings* with compensation code. Code transformations become OSR-aware in isolation, and can then be combined in arbitrary ways.

3.1 OSR Mappings

An *OSR mapping* encodes the machinery required to transfer execution from a point l in a program P to a point l' in a program P′, relying on a compensation code C to fix the memory store so that execution can safely continue in P′. Ideally, this should be done for any realizable state s of P, i.e., s is reachable by P from some initial store $\hat{\sigma}$, to a realizable state s' of P′. For each point l where the mapping is defined, C computes the expected values for live variables at the OSR landing point l' in P′ from the live store at l in P. We formalize these concepts in the following definition:

Definition 3.1 (OSR Mapping)**.** For any $\mathrm{P}, \mathrm{P}' \in \mathit{Prog}$, an *OSR mapping* from P to P′ is a (possibly partial) function $\mathrm{M}_{\mathrm{PP}'} : [1, |\mathrm{P}|] \to [1, |\mathrm{P}'|] \times \mathit{Prog}$ such that:

$$\forall \hat{\sigma} \in \Sigma, \forall s = (\sigma, l) \in \tau_{\mathrm{P}\hat{\sigma}} : l \in \mathit{dom}(\mathrm{M}_{\mathrm{PP}'}),$$
$$\exists \hat{\sigma}' \in \Sigma, \exists s' = (\sigma', l') \in \tau_{\mathrm{P}'\hat{\sigma}'} :$$
$$\mathrm{M}_{\mathrm{PP}'}(l) = (l', \mathrm{C}) \ \wedge\ [\![\mathrm{C}]\!](\sigma)|_{\mathtt{live}(\mathrm{P}', l')} = \sigma'|_{\mathtt{live}(\mathrm{P}', l')}$$

We say that the mapping is *strict* if $\hat{\sigma}' = \hat{\sigma}$.

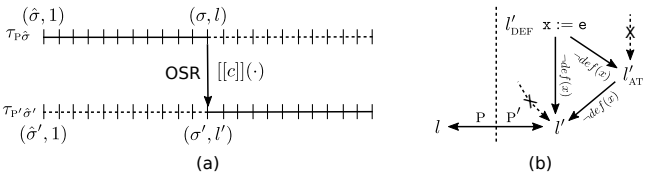

Figure 4. (a) OSR mapping allowing a transition from program P at l to program P′ at l'. (b) Algorithm `reconstruct` identifies an assignment `x := e` at l'_{DEF} that reaches both l' and l'_{AT}, and no other definition of `x` is possible.

The scenario of Definition 3.1 is illustrated in Figure 4(a). Note that we use traces to describe feasible execution states for the two programs.

If P′ has been derived from P via a code transformation T, we say that T can be made *OSR-aware* if we can build a forward mapping $\mathrm{M}_{\mathrm{PP}'}$ and a backward mapping $\mathrm{M}_{\mathrm{P}'\mathrm{P}}$. When a mapping is partial, there are points where OSR is not supported.

To simplify the discussion, we have used a weak notion of store equality restricted to live variables, leaving `load`/`store` instructions for discussion in Section 5.3.

The correctness of fixing only live variables in an OSR mapping stems from the following claim, which states that replacing at any time a store with another one where we only keep live variables does not lead to a different output for the program:

Theorem 3.2. *Given any program $P \in \mathit{Prog}$ and any initial store $\hat{\sigma} \in \Sigma$, it holds that for any state $(\sigma, l) \in \tau_{P\hat{\sigma}}$:*

$$(\sigma, l) \Rightarrow_P^* (\sigma', |P|) \iff (\sigma|_{\mathtt{live}(P,l)}, l) \Rightarrow_P^* (\sigma'', |P|)$$

where $\sigma'|_{\mathtt{live}(P,|P|)} = \sigma''|_{\mathtt{live}(P,|P|)}$ specifies the content of the output variables of the program.

Proof (Sketch). To prove the claim, it is sufficient to reason inductively on the number of transitions required to reach the final instruction at program point |P| using the following argument. For any state $(\sigma_1, l_1) \in \tau_{\mathrm{P}\hat{\sigma}}$, it holds:

$$(\sigma_1, l_1) \Rightarrow_{\mathrm{P}} (\sigma_2, l_2) \iff (\sigma_1|_{\mathtt{live}(\mathrm{P},l_1)}, l_1) \Rightarrow_{\mathrm{P}} (\sigma'_2, l_2)$$

where $\sigma_2|_{\mathtt{live}(\mathrm{P},l_2)} = \sigma'_2|_{\mathtt{live}(\mathrm{P},l_2)}$, that is σ_2 and σ'_2 have the same live variables at l_2. This follows by noticing that, if I_{l_1} kills/creates a live variable when executed on (σ_1, l_1), then it does the same when executed on $(\sigma_1|_{\mathtt{live}(\mathrm{P},l_1)}, l_1)$. Also, if an expression is evaluated by I_{l_1}, then all variables it contains must be live, otherwise we would be contradicting the definition of liveness. Hence, they are defined with the same value in both σ_1 and $\sigma_1|_{\mathtt{live}(\mathrm{P},l_1)}$, leading to the same effect of I_{l_1}. Therefore, the resulting stores σ_2 and σ'_2 concide on the set of live variables of P at l_2. □

Theorem 3.2, applied to P′, allows concluding that an OSR transition from P to P′ leads to a state from which we get the same output we would have obtained by running P′ instead of P.

3.2 OSR Mapping Composition

A relevant property of our framework is that OSR mappings can be composed. This allows us to make transformation OSR-aware in isolation and apply them in sequence, as their compensation code is amenable to program composition.

Definition 3.3 (Program composition)**.** We say that two programs $\mathtt{P} = \langle I_1, \ldots, I_n \rangle$ and $\mathtt{P}' = \langle I'_1, \ldots, I'_{n'} \rangle$ are *composable* if $I_n = \mathtt{out}\ x_1, \ldots, x_k$ and $I'_1 = \mathtt{in}\ x'_1, \ldots, x'_{k'}$ with $\{x'_1, \ldots, x'_{k'}\} \subseteq \{x_1, \ldots, x_k\}$. For any pair of composable programs $\mathtt{P}, \mathtt{P}'$, we define $\mathtt{P} \circ \mathtt{P}' = \langle I_1, \ldots, I_{n-1}, \hat{I}'_2, \ldots, \hat{I}'_{n'} \rangle$, where $\forall i \in [1, n']$, $\hat{I}'_i$ is obtained from I'_i by relocating each `goto` target m with $m + n - 2$.

The semantic function $[\![\mathtt{P} \circ \mathtt{P}']\!](\sigma)$ is defined as $[\![\mathtt{P}']\!]([\![\mathtt{P}]\!](\sigma))$ for any input store σ. Mapping composition is as follows:

Theorem 3.4 (Mapping Composition)**.** *Let $\mathtt{P}, \mathtt{P}', \mathtt{P}'' \in Prog$, let $M_{\mathtt{P}\mathtt{P}'}$ and $M_{\mathtt{P}'\mathtt{P}''}$ be OSR mappings as in Definition 3.1, and let $M_{\mathtt{P}\mathtt{P}'} \circ M_{\mathtt{P}'\mathtt{P}''}$ be a* composition of mappings *defined as follows:*

$$\forall l \in dom(M_{\mathtt{P}\mathtt{P}'}) : M_{\mathtt{P}\mathtt{P}'}(l) = (l', c) \wedge l' \in dom(M_{\mathtt{P}'\mathtt{P}''}) :$$
$$M_{\mathtt{P}'\mathtt{P}''}(l') = (l'', c') \implies (M_{\mathtt{P}\mathtt{P}'} \circ M_{\mathtt{P}'\mathtt{P}''})(l) = (l'', c \circ c')$$

Then $M_{\mathtt{P}\mathtt{P}'} \circ M_{\mathtt{P}'\mathtt{P}''}$ is an OSR mapping from $\mathtt{P}$ to $\mathtt{P}''$.

The theorem allows us to flexibly combine transformation rules, provided that an OSR mapping between the original and modified programs can be produced for each rule.

3.3 Discussion

Our framework aims at supporting classic and also speculative optimizations, making them OSR-aware in isolation. Observe that while strict mappings are a natural fit for semantics-preserving transformations, Definition 3.1 is rather general, as a non-strict mapping may capture transitions to a program $\mathtt{P}'$ that is *not* semantically equivalent to $\mathtt{P}$. For instance, $\mathtt{P}'$ may contain speculatively optimized code, or just some optimized fragments of $\mathtt{P}$ as in tracing JIT [3, 17, 18]. Execution in $\mathtt{P}'$ can then be invalidated by firing an OSR back to $\mathtt{P}$ or to some other recovery program.

Minor modifications are required to include function calls and memory accesses, which we support already in the implementation. However, one limitation is that the current formalization would not allow an executing function to trigger an OSR for its caller: we plan to address frame replacement in future work. Also, we wish to add support for optimizations such as software pipelining [26] where the OSR landing location depends on the run-time values of variables.

4 Automating OSR Mapping Generation

In this section we describe how to automatically construct OSR mappings between program versions generated using compiler transformations T that satisfy a natural property that we call *live-variable equivalence*.

Constant propagation (CP)
m : `x := e[v]` $\implies$ `x := e[c]`
`if conlit(c)` $\wedge$ $m \models \overleftarrow{A}(\neg \mathtt{def(v)}\ U\ \mathtt{stmt(v := c)})$
Dead code elimination (DCE)
m : `x := e` $\implies$ `skip`
`if` $m \models \overrightarrow{AX}\ \neg \overrightarrow{E}(true\ U\ \mathtt{use(x)})$
Code hoisting (Hoist)
p : `skip` $\implies$ `x := e`
q : `x := e` $\implies$ `skip`
`if` $p \models \overrightarrow{A}(\neg \mathtt{use(x)}\ U\ \mathtt{point}(q)) \wedge$
$\quad q \models \overleftarrow{A}((\neg \mathtt{def(x)} \vee \mathtt{point}(q)) \wedge \mathtt{trans}(e)\ U\ \mathtt{point}(p))$

Figure 5. Rewrite rules for LVE common transformations.

4.1 Live-Variable Equivalent Transformations

Intuitively, live-variable equivalence of a transformation T states that the live variables a program $\mathtt{P}$ and its transformed version $\mathtt{P}' = \lceil T \rceil(\mathtt{P})$ have in common at any corresponding state hold the same values. By corresponding states we mean those that would be reached at any time by running simultaneously the two programs from the same initial store. To characterize this property more precisely and prove that a transformation is live-variable equivalent, we need to introduce some formal machinery based on bisimulation.

Definition 4.1 (Program Bisimulation)**.** A relation $R \subseteq State \times State$ is a bisimulation relation between two programs $\mathtt{P}$ and $\mathtt{P}'$ if for any input store $\sigma \in \Sigma$ it holds:

$$s \in \tau_{\mathtt{P}\sigma} \wedge s' \in \tau_{\mathtt{P}'\sigma} \wedge s\ R\ s' \implies$$
$$1)\ s \Rightarrow_{\mathtt{P}} s_1 \implies s' \Rightarrow_{\mathtt{P}'} s'_1 \wedge s_1\ R\ s'_1$$
$$2)\ s' \Rightarrow_{\mathtt{P}'} s'_1 \implies s \Rightarrow_{\mathtt{P}} s_1 \wedge s_1\ R\ s'_1$$

Our notion of bisimulation between programs $\mathtt{P}$ and $\mathtt{P}'$ requires that R be a bisimulation between transition systems $(\tau_{\mathtt{P}\sigma}, \Rightarrow_{\mathtt{P}})$ and $(\tau_{\mathtt{P}'\sigma}, \Rightarrow_{\mathtt{P}'})$ for any store $\sigma \in \Sigma$. This implies that for any σ, $\tau_{\mathtt{P}\sigma}$ is finite iff $\tau_{\mathtt{P}'\sigma}$ is finite; also, if they are finite, then they have the same length. This assumption can be made without loss of generality, as equal length of traces can be enforced by padding programs with `skip` statements.

As a second ingredient, we need to characterize relations between states that tolerate differences, as those that we would get by comparing the corresponding states of two different, but semantically equivalent, program versions:

Definition 4.2 (Partial State Equivalence)**.** For any function $A : \mathbb{N} \to 2^{Var}$, the *partial state equivalence* relation $R_A \subseteq State \times State$ is defined as:

$$R_A \triangleq \{(s, s') : s = (\sigma, l) \wedge s' = (\sigma', l) \wedge \sigma|_{A(l)} = \sigma'|_{A(l)}\}$$

Using this notion, we say that two programs $\mathtt{P}$ and $\mathtt{P}'$ are live-variable bisimilar if for any initial store they hold at any time the same values for variables that are live in both programs:

Definition 4.3 (Live-Variable Bisimilar Programs)**.** $\mathtt{P}$ and $\mathtt{P}'$ are *live-variable bisimilar* (LVB) if R_A is a bisimulation relation between them, where $A = l \mapsto \mathtt{live}(\mathtt{P}, l) \cap \mathtt{live}(\mathtt{P}', l)$

is the function that yields for each program point l the set of variables that are live at l in both P and P′.

Finally, we characterize live-variable equivalent transformations as those that yield LVB programs:

Definition 4.4 (Live-Variable Equivalent Transformation)**.** A program transformation T is *live-variable equivalent* (LVE) if for any program P, P and $\lceil T \rceil$(P) are live-variable bisimilar.

Live-variable equivalence is a natural property of fundamental compiler optimizations that insert, delete, or move instructions around. We report in Figure 5 three examples of semantics-preserving (see [27, 28]) LVE transformations.

Theorem 4.5. *Transformations CP, DCE, and Hoist of Figure 5 are live-variable equivalent.*

The argument for the proof follows the bisimulation relations used in [27] to prove them correct. For CP, R is simply the identity relation, while for DCE and Hoist it is piecewise-defined on the indexes of the traces. Notice that Hoist expects a `skip` to exist at the point where an instruction is moved.

4.2 Mapping Generation

We now discuss how to enhance an existing LVE transformation so that forward and backward OSR mappings for the rewritten program can be generated automatically. We propose a general `OSR_trans`(P, T)→(P′, $\mathrm{M}_{\mathrm{PP'}}$, $\mathrm{M}_{\mathrm{P'P}}$) algorithm that for an input program P and an LVE transformation T builds:

1. a program P′ = $\lceil T \rceil$(P);
2. a forward OSR mapping $\mathrm{M}_{\mathrm{PP'}}$ from P to P′;
3. a backward OSR mapping $\mathrm{M}_{\mathrm{P'P}}$ from P′ to P.

The algorithm relies on two subroutines:

1. `apply`(P, T) → (P′, $\Delta_{\mathrm{PP'}}$, $\Delta_{\mathrm{P'P}}$)
   Builds a program P′ by applying a transformation T on program P, and two functions $\Delta_{\mathrm{PP'}}$, $\Delta_{\mathrm{P'P}}$ that map program points between P and P′.
2. `reconstruct`(x, P, l, P′, l', l'_{AT}) → C
   Takes a variable x, the OSR origin and destination points l and l' in programs P and P′, respectively, and an additional point l'_{AT} in P′. It builds a compensation code fragment C that assigns x with the value it would have had at l'_{AT} just before reaching l', had execution been carried on in P′ instead of P. This scenario is illustrated in Figure 4(b).

The workflow of `OSR_trans` is as follows. To construct an OSR mapping $\mathrm{M}_{\mathrm{PP'}}$ from P to P′, it attempts to build a compensation code C for each ordered pair (l, l') of program points in the $\Delta_{\mathrm{PP'}}$ function returned by `apply`(P, T). C is a program that takes as input the $x_1, \ldots, x_k$ live variables for the OSR source location l in P, and assigns the correct values to the $x'_1, \ldots, x'_{k'}$ live variables for execution to resume at l' in P′.

When dealing with an LVE transformation, `reconstruct` needs only to be invoked for variables that are live at the OSR destination but not at the source. The LVB hypothesis for P and P′ entails that for any pair of corresponding locations, variables that are live in both programs hold the same value. We exploit this property in the formulation of `reconstruct` given in Algorithm 1 for LVE transformations.

ALGORITHM 1: Value reconstruction for LVE programs.

```
let ud(x, P̄, l_D, l_R) ≜ P̄, l_R ⊨ ←AX←A(¬def(x) U point(l_D) ∧ def(x))
procedure reconstruct(x, P, l, P′, l′, l′_AT)
1  if ∃ l′_DEF : ud(x, P′, l′_DEF, l′_AT)
2      if l′_DEF is visited return ⟨⟩
3      mark l′_DEF as visited
4      if ud(x, P, l′_DEF, l′) ∧ x ∈ live(P′, l′) ∩ live(P, l) return ⟨⟩
5      C ← ⟨⟩
6      foreach y : y ∈ freevar(e) ∧ P′, l′_DEF ⊨ stmt(x:=e)
7          C ← C · reconstruct(y, P, l, P′, l′, l′_DEF)
8      return C ← C · x:=e
9  else throw undef
```

Algorithm description. `reconstruct` is called with $l'_{\mathrm{AT}} = l'$ by `OSR_trans`. The algorithm first checks[1] whether there is a unique reaching definition for x at some point l'_{DEF} in P′ reaching l'_{AT}. In the presence of multiple reaching definitions, the algorithm gives up. If x is live both at the origin l and at the destination l', and the definition of x at l'_{DEF} that reaches l'_{AT} is also a unique reaching definition for x at l' (line 4), then x would have assumed at l'_{AT} the same value available at l'. For the LVB hypothesis, if x is already available at the origin no value reconstruction is needed (`return` at line 4).

If x is not available at l, `reconstruct` iterates over all the constituents of the assignment x := e computed at l'_{DEF}, i.e., the variables that occur in e, and recursively builds code to compute the values that they would have assumed at l'_{DEF} just before reaching l' if execution had been carried on in P′. Once the recursively generated code has been added to C, the assignment x := e is appended to it. Note that we mark program points to avoid work repetition.

Construction of the $\mathrm{M}_{\mathrm{P'P}}$ mapping is analogous up to index renaming, thus details are not reported. The LVB hypothesis is a sufficient condition for the correctness of our approach:

Theorem 4.6. *For any program P and LVE transformation T, if* `apply`$(P, T) \triangleq (P', \Delta^I, \Delta^I)$ *where* $\Delta^I : [1, |P|] \to [1, |P|]$ *is the identity mapping between program points, then* `OSR_trans` $(P, T) \to (P', M_{PP'}, M_{P'P})$ *yields strict forward and backward OSR mappings* $M_{PP'}$ *and* $M_{P'P}$*, respectively, for programs P and P′.*

Discussion. Theorem 3.4 provides us with the flexibility to compose mappings generated from the LVE formulation given in Algorithm 1 for `reconstruct` with mappings from other transformations. Hence, it would be interesting to explore variants of the algorithm tailored to other classes of optimizations (e.g., vectorization-based ones) as well.

[1] Predicate ud(x, P̄, l_{D}, l_{R}) captures the existence in a program P̄ of a unique definition, located at l_{D}, for variable x that reaches location l_{R}.

Algorithm 1 currently gives up when analyzing a variable with multiple reaching definitions. Compilers using a static single assignment (SSA) representation [12] model such a variable using a ϕ function. *Gating functions* [36] might offer a solution to us, as they capture the control conditions that determine which incoming definition for a ϕ function will provide the value. Compensation code could thus evaluate them against the current state and materialize a value accordingly. We plan to explore this direction as future work.

5 Implementation

In this section we first describe the steps required for implementors to accommodate our ideas in a general-purpose compiler, then we present details for an embodiment in LLVM.

5.1 Mapping Program Points and Variables

In our abstract model, we implicitly map variables by the same name, and unrealistically assume that corresponding states for LVB programs are located at the same points.

In practice, OSR embodiments and deoptimization handlers typically use scope descriptors to track program points and local variables [45]. When such metadata are not available, one can instrument optimizations at places where IR manipulations happen. We argue that for LVE transformations, it is enough to track the following primitive actions:

1. `add`(*inst*, *loc*): insert a new instruction at *loc*
2. `delete`(*loc*): delete instruction at *loc*
3. `hoist`(*loc*, *newLoc*): hoist instr. from *loc* to *newLoc*
4. `sink`(*loc*, *newLoc*): sink instr. from *loc* to *newLoc*
5. `replace`(*oldOp*, *newOp*, [*inst*]): uses of an operand in *inst* or in the entire code are replaced with another

This comes at a little cost in some environments: for instance, the profiling technique described in [51] relies on similar actions (e.g., node elimination and movement, value node replacement) that are exposed by the Graal compiler as IR graph manipulations. Others such as LLVM require instead manual placement of hooks inside the transformations, regardless of where one wants to place an OSR point. Once the tracking is done, no further changes to LVE optimizations are required to support OSR.

5.2 Reconstructing Missing Variables

Once optimizations have been applied and IR mappings created by `apply`, compensation code can be automatically built using `reconstruct`. When an SSA representation is used, implementing Algorithm 1 is easier, as the reaching definition for a variable is unique at any point it dominates.

In general, a code optimizer might decide to keep a variable alive to support deoptimization at some location, provided that the optimized version can still execute faster. In a similar spirit, we devise two variants of `reconstruct`: a *live* version using only live variables at the OSR source, and an *avail* version that keeps alive already-computed values that the algorithm cannot reconstruct otherwise. Keeping a variable x alive should not be a concern in terms of register pressure increase, as registers are occupied for the minimum necessary: when x is assigned to a register, a compiler spills it when about to be clobbered, to only reload it later when an OSR is about to happen; when assigned to a stack location instead, x is loaded to a physical register only at OSR time.

5.3 Supporting Memory Manipulation

Compilers rely on `load` and `store` operations to transfer values between memory and registers. A simple sufficient condition for correctness of transitions is that `store` instructions are executed at the same program point in all versions. Indeed, when two program versions assign to a variable with a `load` from the same address, and the variable is live at some same program point in both versions, then the value read from memory has to be the same in both versions. Our implementation preserves the `store` invariant above while allowing instructions that do not access memory to be hoisted above or sunk below a `store` instruction. Notice that common LLVM optimizations such as loop hoisting and code sinking deal with `store` instructions in a similar manner.

A possible extension for scenarios where the above assumption might be too restrictive is as follows. Suppose that a `store` is sunk during optimization. For each CFG location between the original location and the insertion point: (a) in an optimizing OSR, no compensation code is required, as the `store` has been executed already, and re-executing it at the insertion point will be harmless; (b) in a deoptimizing OSR, we realign the memory state by executing the sunk `store`, which has not been reached yet in the optimized version.

5.4 LLVM Implementation

We conclude the section by reporting implementation details specific to LLVM. For the reader not familiar with it, optimizations are expressed as *passes* that manipulate IR in SSA form and are shared by front-ends for a variety of languages. Architecture-specific optimizations are performed instead in the back-end. For simplicity, front-ends are allowed to place variables on the stack: the `mem2reg` pass will then promote stack references to registers in SSA form.

Compared to our abstract model, IR conditionals and assignment instructions uniquely determine program locations, while virtual registers correspond to variables in the store.

Making Passes OSR-Aware. Our implementation of `apply` takes as input a function and a sequence of optimizations, clones the function, optimizes the clone, and eventually constructs a mapping for program points and local variables between the two versions by processing the history of applied actions (Section 5.1). IR mapping construction is implemented in ~300 C++ LOC.

Table 1. Edits performed to original LLVM passes.

	ADCE	CP	CSE	LICM	SCCP	Sink	LC	LCSSA	other
LOC	64	60	423	413	987	167	521	191	613
changed	4	5	16	13	12	6	45	17	31
actions	1	2	10	8	4	1	11	3	13

We make OSR-aware a number of standard optimizations, including aggressive dead code elimination (ADCE), constant propagation (CP), common subexpression elimination (CSE), loop-invariant code motion (LICM), sparse conditional constant propagation (SCCP), and code sinking (Sink). We also augment the natural loop canonicalization (LC) and the loop-closed SSA form-construction (LCSSA) utility passes required by LICM. Optimizations performed in the back-end such as instruction scheduling and register allocation do not require instrumentation, as we operate at IR level.

```
if (isInstructionTriviallyDead(Inst, ...)) {
  OSR_CM->deleteInstruction(Inst);
  Inst->eraseFromParent();
  Changed = true; continue; }

if (SimpleValue::canHandle(Inst)) { // value number
  if (Value *V = AvailableValues->lookup(Inst)) {
    OSR_CM->replaceAllUsesWith(Inst, V);
    OSR_CM->deleteInstruction(Inst);
    Inst->replaceAllUsesWith(V);
    Inst->eraseFromParent();
    Changed = true; continue; } [...]

if (LoadInst *LI = dyn_cast<LoadInst>(Inst)) { [...]
  // check for available load from right generation
  std::pair<Value*, unsigned> InV =
    AvailableLoads->lookup(Inst->getOperand(0));
  if (InV.first != nullptr && InV.second == CurGen) {
    if (!Inst->use_empty()) {
      OSR_CM->replaceAllUsesWith(Inst, InV.first);
      Inst->replaceAllUsesWith(InV.first); }
    OSR_CM->deleteInstruction(Inst);
    Inst->eraseFromParent();
    Changed = true; continue; } [...] }
```

Figure 6. Excerpt of instruction processing block in CSE. An `OSR_CM` mapper object is used to track primitive actions.

Table 1 reports figures for our edits to the original code, along with the number of primitive actions we track in each pass. Changes involve updating a `CodeMapper` object (which tracks IR updates) by instrumenting points where manipulations happen. The last column describes changes to utility methods that are shared between passes and that we analyze only once. Figure 6 shows an excerpt of an OSR-aware pass: actions (Section 5.1) often mimic common utilities for LLVM IR manipulation, thus are simple to place once one has an understanding of the high-level behavior of a pass.

`reconstruct` can take advantage of peculiarities of LLVM IR, too. For instance, we identify and reconstruct ϕ-nodes that always evaluate to the same value, such as those artificially inserted when constructing the LCSSA form. We also capture implicit aliasing information from `replace` actions.

OSR Transitions. LLVM is used as a dynamic compiler in several projects. The OSRKit library [13] provides the machinery to devise an OSR transition between two program points, requiring the VM's front-end to provide a mapping between virtual registers in the two versions and the state realignment code possibly required for the transition. [13] tackles the engineering aspects for supporting OSR in LLVM, presenting a case study in which glue code is hand-written to support type specialization via dynamic inlining in MATLAB. We hopefully make a step forward by showing for LVE transformations how to automatically generate glue code (and IR mappings) and compose them on top of OSRKit.

OSRKit can insert an OSR point in a function f at a location l to a variant f', guarding it with a user-provided condition. The transition is modeled as a call that transfers the live state to a *continuation function* f'_{to} that can be generated ahead of time or on the fly. f'_{to} will execute any compensation code c placed in its entry block before jumping to the resumption point l'. As OSRKit works at IR level, we provide it with a mapping between live IR registers at the OSR point, and a sequence c of instructions that can materialize values that are live in f' but not in f. As OSR metadata, we only maintain the `CodeMapper` for the IR objects of the two code versions, as glue code is generated on demand by `reconstruct`.

Note that as f'_{to} is a specialization of f' having l' as unique entry point, it can typically execute faster than the original f' [15]. In our case, deleting unreachable blocks yields more compact code, possibly improving register allocation, too. Also, f'_{to} in principle can be further optimized, including c.

Due to its placement, c does not affect optimization decisions nor the steady-state overhead from the presence of OSR points, which is minimal for OSRKit [13]. c is executed only once and is typically small in practice (Table 3). OSR guards are transparent to our framework, as we reason about uninstrumented code variants. To extend the liveness range of a virtual register for *avail*, it is sufficient to add it as argument for the call to f'_{to}. An optional OSR edge probability can be used to refine native code generation.

6 Evaluation

A preliminary investigation on classic benchmarks suggests that our techniques may enable bidirectional OSR almost everywhere in the face of common code optimizations.

6.1 Benchmarks and Environment

We integrate our techniques in the TinyVM artifact[2] from the OSRKit paper [13] for IR optimization, JIT compilation, and benchmarking. Construction and composition of OSR

[2]Additions are available at https://github.com/dcdelia/tinyvm.

Table 2. IR features of analyzed code. We report the number of instructions |f| (|ϕ| of which are ϕ-nodes) in f_{base} and f_{opt}, and primitive actions tracked during optimization.

Benchmark	$\lvert f_{base} \rvert$	$\lvert \phi_{base} \rvert$	$\lvert f_{opt} \rvert$	$\lvert \phi_{opt} \rvert$	add	delete	hoist	sink	replace
bzip2	657	32	596	44	16	77	12	3	73
h264ref	671	28	576	36	9	105	4	21	102
hmmer	568	6	383	8	2	187	13	1	187
namd	1737	159	1636	224	68	169	36	73	162
perlbench	5574	305	5001	355	86	667	96	28	627
sjeng	1940	93	1540	105	13	413	20	34	413
soplex	195	2	154	2	0	41	2	4	41
bullet	587	24	553	42	26	60	37	3	52
dcraw	590	37	545	49	13	58	25	6	58
ffmpeg	618	34	462	40	11	168	9	17	103
fhourstones	288	29	284	39	14	20	3	0	16
vp8	334	41	299	60	19	54	17	34	54

mappings takes place when a sequence of passes is applied to a function to obtain an optimized variant of it.

We evaluate our technique on the SPEC CPU2006 [21] and the Phoronix PTS [38] benchmarking suites, reporting data for a subset of their C/C++ benchmarks. We profile the hottest method in each benchmark and when it accounts for at least 5% of the execution time, we generate LLVM IR for an f_{base} version using `clang` with the sole `mem2reg` pass enabled. We then produce an f_{opt} version by applying the optimizations discussed in Section 5.4.

To validate our implementation, we compile and run the code we generate for a sample of all feasible OSR source-destination pairs. We use a machine equipped with an Intel Core i7-3632QM, Ubuntu 14.10 64-bit and LLVM 3.6.2.

6.2 Results

In Table 2 we report aggregate figures for IR manipulations performed by the optimizations. While the f_{opt} version is typically shorter than its f_{base} counterpart, it might have a larger number of ϕ-nodes: most extra nodes are commonly generated during the LCSSA-form construction and optimized away in the back-end. SCCP can eliminate a large number of unreachable blocks from `ffmpeg`, while CSE performs the majority of deletions in the other benchmarks.

Optimizing OSR. Figure 7 shows how many of the program points are feasible for an OSR from f_{base} to f_{opt} depending on the version of `reconstruct` in use. Locations that do not need a compensation code for OSR, i.e., $c = \langle\rangle$, account for a limited fraction of all the program points (less than 10% for most benchmarks), suggesting that optimizations can significantly affect how the live state of a program looks.

We observe that *live* performs well on most benchmarks: for 9 out of 12 programs, we can build a compensation code using only live variables at the OSR source for more than 60% of the program points. For *avail* the percentage of feasible

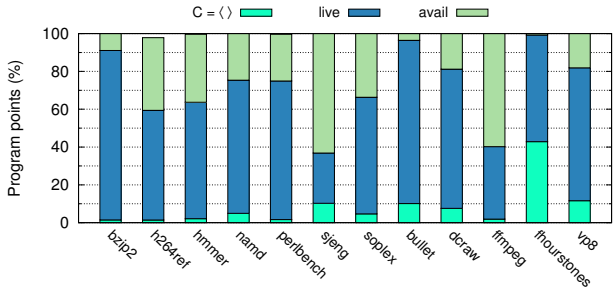

Figure 7. Breakdown of feasible $f_{opt} \rightarrow f_{base}$ OSR points.

OSR points grows almost to 100% for all benchmarks. The ability to identify ϕ-nodes yielding a constant value is crucial to support OSR at ~20% of the program points in `bullet`.

In Table 3 we report average and peak size of the compensation code c generated by *live* and *avail* for feasible OSR points. Notice that averages are calculated on different sets of program points, i.e., *avail* extends the set from *live*. The assignment step of Algorithm 1 (line 8) generates an average number of instructions typically smaller than 20, with the notable exception of `perlbench`. Its hottest function highly benefits from CSE: no less than 583 out of its 667 deleted instructions (~10% of the size of f_{base}) are removed by it. Local CSE may reduce the size of the OSR entry block, too. However, we do not optimize c by default as the impact of compensation code on performance is hardly noticeable.

Table 3 also reports average and peak number of variables ($|K_{avail}|$) that are not live at the source location, but *avail* artificially keeps alive to support OSR at more points. Their average number is less than 3 for 9 out of 12 benchmarks, with a maximum of 6.15 for `bullet`. Also, notice that *avail* uses a simple backtracking strategy to identify the minimal set of variables that cannot be reconstructed otherwise.

Deoptimizing OSR. Figure 8 shows how many program points are eligible for f_{opt}-to-f_{base} deoptimization. We observe that the fraction of locations that can fire an OSR with an empty c varies significantly among benchmarks, suggesting a dependence on the structure of the original program.

For 9 out of 12 benchmarks, compensation code can be built using only live variables for more than 50% of potential OSR points. When the *avail* version is used, the percentage of OSR-feasible program points is greater than 90% on all benchmarks and nearly 100% for 9 out of 12 of them.

In Table 3 we report statistics on the size of the compensation code generated across feasible OSR points and the number of variables to be kept alive by *avail*. Compared to the optimizing OSR scenario, the size of c is much smaller, suggesting that shorter portions of execution need to be reconstructed in a deoptimizing OSR. Although we report a 0 size for `fhourstones`, some realignment is still needed: our code contains minor optimizations not discussed here that can detect when there is a live alias for a variable x that can be used in its place throughout the IR.

Table 3. Average and peak size |c| of the compensation code generated by the two versions of `reconstruct`. $|K_{avail}|$ is the size of the set of variables to be kept artificially alive for OSR at program points represented by the top bars in Figure 7 and 8.

	$f_{base} \rightarrow f_{opt}$						$f_{opt} \rightarrow f_{base}$					
	$\|c\| \leftarrow live$		$\|c\| \leftarrow avail$		$\|K_{avail}\|$		$\|c\| \leftarrow live$		$\|c\| \leftarrow avail$		$\|K_{avail}\|$	
Benchmark	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
bzip2	4.3	14	4.73	13	3.6	8	1.55	4	1.77	4	1.47	4
h264ref	2.9	5	3.37	5	1.02	2	4.46	9	2.82	9	1.45	7
hmmer	16.11	23	16.63	24	4.02	7	1	1	1	1	1.02	2
namd	18.61	28	17.82	28	3.38	6	1.5	2	5.93	15	4.74	18
perlbench	46.12	57	45.82	57	1.24	12	4.09	12	4.22	12	1.37	11
sjeng	9.72	21	18.52	32	4.2	12	1.29	2	1.67	11	4.09	14
soplex	5.02	7	4.38	7	2.34	4	3.3	4	3.3	4	1.00	1
bullet	16.69	46	15.93	46	6.15	17	1	1	1.26	3	1.14	2
dcraw	7.6	15	7.32	15	1.97	7	1.68	2	3.84	6	4.06	8
ffmpeg	5.05	8	4.03	8	1.85	3	1.94	5	1.95	6	1.08	4
fhourstones	4.5	6	4.98	6	1.7	2	0	0	1.12	4	1.42	4
vp8	10.51	16	10.13	17	2.35	6	5.74	13	5.51	13	1.18	5
Avg	**12.26**	20.50	**12.81**	21.50	**2.82**	7.17	**2.30**	4.58	**2.87**	7.33	**2.00**	6.67

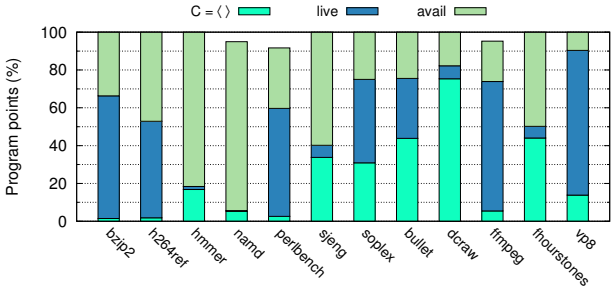

Figure 8. Breakdown of feasible $f_{base} \rightarrow f_{opt}$ OSR points.

7 Symbolic Debugging of Optimized Code

In this section we present a feasibility study of how our algorithms for compensation code generation can provide useful novel building blocks for optimized-code debuggers. On prominent C benchmarks, `reconstruct` is able to recover the expected source-level values for the vast majority of scalar user variables that might not be reported correctly by a debugger due to the effects of classic compiler optimizations.

7.1 Background

A *source-level* (or *symbolic*) debugger allows a programmer to monitor an executing program at the source-language level. Interactive mechanisms are typically provided to the user to halt/resume the execution at breakpoints, and to inspect the state of the program in terms of its source language.

The importance of the design and use of these tools was already clear in the '60s [14]. Optimizations are desirable in a production environment, and bugs can surface when they are enabled: a debuggable translation may hide them, or differences in timing behavior may trigger race conditions. Also, optimization may be mandatory due to memory limitations, efficiency reasons, or other platform-specific constraints [1].

Hennessy [20] describes the classic conflict between the use of optimizations and the ability to debug a program symbolically. A debugger provides the user with the illusion that the source program is executing one statement at a time. Optimizations preserve semantic equivalence, but can alter the structure and the intermediate results of the program.

Two problems surface when trying to symbolically debug optimized code [2, 25]: i) the *code location* problem, as the debugger must determine the position of a breakpoint in the optimized code, and ii) the *data location* problem, as users expect values of variables at a breakpoint to be consistent with the source code, but instructions may have been deleted or reordered, or values been "lost" due to register allocation.

When attempting to debug optimized programs, debuggers may thus give misleading information about the value of variables at breakpoints. Hence, the programmer has the difficult task of attempting to unravel the optimized code and determine what values the variables should have [20]. When global optimizations can cause the run-time value of a variable to be inconsistent with the source-level value expected at the breakpoint, the variable is called *endangered* [2].

A symbolic debugger has two ways of presenting meaningful information for an optimized program [47]: a) it can provide *expected behavior* by hiding the effects of optimizations, presenting the program state consistently with what the user expects from the source; or b) it can provide *truthful behavior* by making them aware of the effects of the optimizations, and warning them of possibly surprising outcomes.

Constraining optimizations or adding extra code during compilation to aid debugging does not solve the problem of debugging the optimized translation of a program, as the user debugs suboptimal code [1]. Source-level debuggers should thus explore techniques to recover expected behavior without relying on intrusive compiler extensions.

7.2 Using `reconstruct` for State Recovery

Dynamic deoptimization was pioneered in the SELF VM to provide expected behavior with globally optimized code [23].

Table 4. SPEC CPU2006 C benchmarks suite: for endangered functions, we report weighted Avg_g and unweighted Avg_u average of the fraction of program points with endangered user variables, then mean, standard deviation, and peak number of endangered variables at such points. We use the number of IR instructions $|f_{base}|$ as weight for Avg_w, and consider only IR program points corresponding to source locations.

				Endangered functions				
				Fraction of affected points		End. user vars per aff. point		
	Functions							
Benchmark	$\|F_{tot}\|$	$\|F_{opt}\|$	$\|F_{end}\|$	Avg_w	Avg_u	Avg	σ	Max
bzip2	100	66	24	0.17	0.12	1.22	0.55	5
gcc	5 577	3 884	1 149	0.25	0.22	1.13	0.31	14
gobmk	2 523	1 664	893	0.40	0.29	1.48	0.72	9
h264ref	590	466	163	0.45	0.55	1.69	1.23	14
hmmer	538	429	80	0.17	0.22	1.13	0.37	5
lbm	19	17	2	0.30	0.51	1.97	1.37	3
libquantum	115	85	9	0.13	0.10	1.06	0.17	2
mcf	24	21	11	0.35	0.32	1.00	-	1
milc	235	157	34	0.24	0.21	1.14	0.29	3
perlbench	1 870	1 286	593	0.37	0.35	1.16	0.36	8
sjeng	144	113	31	0.26	0.20	1.24	0.42	3
sphinx3	369	275	76	0.29	0.31	1.19	0.44	6
			Mean	0.26	0.25	1.26	0.47	6.08

Debugging information was supplied by the compiler at discrete *interrupt points* that acted as a barrier for optimizations, letting the compiler run unhindered between them. Motivated by the observation that our algorithms do not limit LVE transformations and can be applied at any location, we investigate whether they can also encode useful information for providing expected behavior in a source-level debugger.

As in most recent works on the topic, we focus on identifying and recovering *scalar source variables* in the presence of global optimizations. LLVM front-ends encode debugging information as metadata attached to IR objects (e.g., instructions, global variables, functions). Metadata are transparent to passes (i.e., they do not affect optimization) and agnostic to both the source language and the target debugging data format. Two intrinsics (`llvm.dbg.value` and `llvm.dbg.declare`) are used to associate source variables with a virtual register or the address of an `alloca` buffer.

We extend TinyVM to rebuild this mapping and identify which locations in the unoptimized IR f_{base} correspond to source-level locations (i.e., possible breakpoints) for a function. An OSR mapping is constructed as well when we apply LVE transformations to generate f_{opt}. For each location in f_{opt} that might correspond to (i.e., have as OSR landing pad) a source-level location in f_{base}, we determine which live variables at the destination are live also at the source (and thus yield the same value), and which ones are instead endangered and may need recovery.

7.3 The SPEC CPU2006 Benchmarks

To capture a variety of programming patterns and styles from applications with different sizes, we analyze each method of each C benchmark from the SPEC CPU2006 suite, applying the OSR-aware optimization passes from Section 5.4 to the baseline IR version f_{base} obtained with `clang -O0` followed by `mem2reg`. Table 4 reports for each benchmark the code size (LOC), the total number of functions in it ($|F_{tot}|$), the number of functions modified by the applied optimizations ($|f_{opt}|$) and, in turn, how many optimized functions are *endangered* ($|F_{end}|$), i.e., contain endangered user variables.

We observe that 11% (`libquantum`) to 54% (`gobmk`) of the optimized functions are endangered, while for 10% to 33% of the functions in each benchmark, the applied passes do not kick in. For endangered functions, on average at more than 25% of program points there is at least a user variable whose source value might not be reported correctly. For most functions in the benchmarks, the average number of affected user variables at such points ranges between 1 and 2, although we sometimes observe higher peaks at specific points (e.g., as high as 9 for `gobmk` and 14 for `gcc` and `h264ref`).

To investigate possible correlations between the size of a function and the number of user variables affected by source-level debugging issues, we analyze the corpus of functions for the three largest benchmarks in our suite, i.e., `gcc`, `gobmk`, and `perlbench`. Our findings suggest that, although larger functions might be more prone to have a large number of affected variables, such issues frequently arise for smaller functions as well.

7.4 Experimental Results

We evaluate the ability of `reconstruct` to recover the source-level expected value for endangered user variables in the SPEC CPU2006 experiments. For each function, we measure the *average recoverability ratio*, defined as the average across all IR points corresponding to source-level locations of the ratio between recoverable and endangered user variables at the point. Two versions of `reconstruct` can be used here.

Implementing `live` should not be difficult for debuggers such as `gdb` and `LLDB` that can evaluate expressions over the current program state, as it only needs to access the live state of the optimized program at the breakpoint. *avail* could use *invisible* breakpoints to spill a number of available values before they are overwritten. Such breakpoints are largely employed in source-level debuggers [25, 47, 50]. Using spilled values and the current live state, expected values for endangered user variables can be reconstructed as for *live*. In a VM with debugging features, one can recompile a function when the user inserts a breakpoint in it, extending the liveness range for available values possibly needed by `reconstruct`.

Figure 9 shows for each benchmark the global average recoverability ratio achieved by *live* and *avail* on the set of affected functions F_{end}. We observe that *avail* performs particularly well on all benchmarks, with a global ratio higher than 95% for half of the benchmarks, and higher than 90% for 10 out of 12 benchmarks. In the worst case (`gobmk`), we observe a global ratio slightly higher than 83%. Results thus

Table 5. Values to be preserved for *avail*. For functions that have at least one such value, we report the fraction $frac$ of $|F_{end}|$ they cumulatively account for, the average number avg of values to preserve across such functions, and the standard deviation σ for it. No liveness extension is required in `lbm`.

	bzip2	gcc	gobmk	h264ref	hmmer	libquantum	mcf	milc	perlbench	sjeng	sphinx3	Mean
$frac$	0.71	0.72	0.16	0.71	0.70	0.67	1.00	0.76	0.66	0.77	0.72	0.69
avg	3.24	2.77	2.31	4.90	2.79	3.00	1.82	2.19	4.76	1.88	2.31	2.91
σ	3.38	5.12	2.22	9.23	2.33	3.46	0.87	1.94	4.94	1.12	2.08	3.34

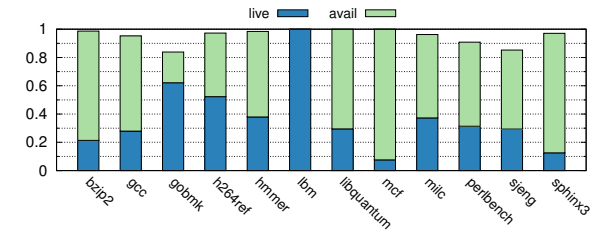

Figure 9. Global average recoverability ratio, defined as the weighted average of each function's average recoverability ratio. As in Table 4, we use $|f_{base}|$ as weight for the average.

suggest that `reconstruct` can recover expected values for the vast majority of source-level endangered variables.

To estimate how many values should be preserved to integrate *avail* in a debugger, we collect for each function the "keep" set of non-live available values to be saved to support deoptimization across all program points corresponding to source locations. We then compute the average and the standard deviation for the size of this set on all the endangered functions. Figures reported in Table 5 show that typically a third of the endangered functions do not require preserving any value. For the remaining functions, 2.91 values need to be preserved on average, with a peak of 4.90 for `h264ref`.

Observe that values in the keep set do not necessarily need to be preserved all simultaneously or at all points, as the minimal set to be maintained may change across function regions. Typically when debugging, values are saved using an invisible breakpoint before they are overwritten, and deleted as soon as they are no longer needed [25].

8 Related Work

On-Stack Replacement. The SELF VM has pioneered dynamic deoptimization for source-level debugging [23] and OSR [24] for efficient dynamic recompilation. The rise of the Java language has then brought OSR technology to the mass market, employing it in the most sophisticated runtimes.

HotSpot Server [37] instruments function entry points and backward branches to optimize performance–critical methods, and transfers execution to the interpreter when an instruction triggers deoptimization, e.g., after class loading.

Jikes RVM places instrumentation as in HotSpot to enable a profile-driven deferred compilation mechanism [15]. OSR enables recovery from speculative inlining too, using a stub to divert execution to a newly generated function. An `OSRBarrier` can be injected when lowering bytecode for interruptible methods to capture the JVM-level program state before the bytecode is executed.

Graal [49] aggressively optimizes code via partial evaluation, tracking the interpreter (i.e., unoptimized) state throughout the compilation: `FrameState` metadata are thus used to reconstruct interpreter frames during deoptimization. [45] proposes an alternate scheme by letting the optimizing compiler generate also the deoptimization target code. This simplifies the implementation of a deoptimization handler, as frames are now described in a uniform format. The Truffle language implementation framework [46] relies on Graal and exposes deoptimization to the programmer, allowing them to transition back to the interpreter and throw away the machine code for a specialization state of a method [48].

We have discussed OSR in LLVM in Section 5.4. Also, an earlier work [29] provides support in the legacy JIT for transitions at loop headers when no state adjustment is required.

V8 uses multiple compilers with the recent addition of an interpreter. The IR graph is processed in an abstract fashion, incrementally tracking changes to program state performed by single instructions. In the lowering phase this information is materialized as deoptimization data where needed. Its highly optimizing TurboFan compiler supports OSR at loop headers, generating a continuation function specialized for the current variable values (as in Jikes) at the loop entry.

Most aforementioned works have an interpreter as deoptimization target and focus on efficiency aspects related to where to support bidirectional OSR transitions between interpreted and compiled code; our goal has been to investigate relations between program points and variables across code versions. We share similarities with [48] in allowing the same compiler for both code versions, and we hope that our algorithms could help [48] support optimizations that make new values alive across deoptimization entry points.

We share the goal of crystallizing OSR abstractions and explore reusable solutions with [43], which prototypes in the Mu micro VM [44] a frame replacement API with an abstract view of stacks that can easily be used in any runtime, and has been implemented even on concrete hardware.

Correctness of Compiler Optimizations. Translation validation [35, 39] aims at verifying semantic equivalence for an optimized translation of a specific program. [27, 28] use rewrite rules with CTL-based side conditions to express transformations and prove them correct, while [30, 31] investigate automated soundness proofs for transformations expressed in a similar manner. Alive [32] has emerged as a domain-specific language for writing sound LLVM peephole optimizations. While these works focus on proving optimizations sound, in this paper we aim at proving OSR correct in the presence of optimizations that we know to be sound.

[26] aims at proving the equivalence of *parameterized programs*, which yields correctness of transformation rules once for all. This approach deserves further investigation in our setting, as it could provide a principled approach to computing mappings between equivalent points in different versions in the face of complex optimizations. [16] studies the correctness of speculative JIT optimizations, showing how incorporating assumptions in the IR makes it easier to reason about equivalence. We look forward to investigating how our techniques and theirs might benefit from each other.

Optimized-Code Debugging. We only know of one work that supports full symbolic debugging with expected behavior. TARDIS [5] is a time-traveling debugger for VMs that saves program state on a regular basis, and runs the unoptimized code in a restored snapshot to answer queries. Our solution is different, as it lies at the performance-preserving end of the spectrum [1], and in some ways more general, as it can be applied to statically compiled languages such as C.

Wu et al. [47] propose a framework to selectively take control of the execution by inserting four kinds of breakpoints, and perform a forward recovery process in an emulator that executes the optimized instructions mimicking their ordering at the source level. The emulation scheme however cannot report values whose reportability is path-sensitive. FULLDOC [25] makes a step further, as it can provide truthful behavior for deleted values, and expected behavior for the other values. The authors remark that FULLDOC can be integrated with techniques for reconstructing deleted values: our `reconstruct` might be an ideal candidate.

Hennessy [20] presents value recovery algorithms for local (with weaker extensions to global) optimizations, but advances in modern compilers make a revision of the assumptions behind them necessary [11]. Adl-Tabatabai [1] presents algorithms that identify aliases for source variables in compiler temporaries introduced by global optimizations. This idea is captured by our technique, which can also use facts recorded during IR manipulation when recursively reconstructing portions of the original program's state.

Other Related Work. Product programs [6] used in verification of relational and k-safety properties are orthogonal to multi-version programs, which embody the notion of OSR and rely on CTL and model checking. [8] discusses loop tiling in the face of `throw` statements that thwart optimizations: to roll back out-of-order updates during deoptimization, an algorithm identifies a minimal number of elements to back up. Our work does not log incremental state changes, and may resort to liveness extension for *available* variables only.

We share similarities with the formalism from [19] for verifiable dynamic software updating (DSU): patches come with heap transformer expressions, while a language construct indicates where updates can take place; symbolic execution [4] and Thor [34] are used for verification. DSU [22] faces more complex requirements than OSR, for instance involving different object versions [33], but can take place once execution reaches a safe point. [41, 42] discuss how to integrate DSU in a VM and devise extensions to the OSR support of Jikes.

Of a different flavor, but in a similar spirit as ours, [18] uses bisimulation to study what trace optimizations are sound. A dynamic compiler uses OSR to replace whole methods; a tracing JIT aggressively optimizes linear recorded instruction sequences, which control flow can leave through guarded side exits only: for instance, RPython [40] uses trampolines to analyze resume information for a guard and runs a compensation code to leave the trace, while SPUR [7] uses a transfer-tail JIT to bridge the execution to the baseline JIT.

9 Conclusions

This paper aims at making a first step towards a more general applicability of OSR. We hope that our ideas can provide the foundation for exploring novel applications of OSR and shed light on its flexibility. Optimizations can significantly affect the live state of a program across its locations: we show how to generate compensation code that runs in O(1) time by recursively reassembling portions of the state for the target function. Preliminary evidence suggests that our techniques can provide useful building blocks for symbolic debuggers. Also, we enable bidirectional transitions at most program locations in a dynamic compilation setting. While we have prototyped them in LLVM, our ideas are general and do not depend on a specific platform or IR representation.

A number of directions for future work are possible. Currently, we only study static transformations that are often enabled by dynamic optimizations: we would like to explore handling purely dynamic ones as well, for instance to materialize objects that have been optimized by scalar replacement. We also plan to model frame replacement and identify other classes of transformations for which a special-purpose replacement can be automatically generated. Heavy-duty transformations may likely require state logging as in the tiling case [8], leaving interesting performance/flexibility trade-offs for exploration. Compensation code with control flow (e.g., using gating functions) might enable state reconstruction at points that our *live* reconstruction algorithm currently does not support. More flexibility might come from observational equivalence as well, by allowing for execution transfers to the last state-changing instruction in lieu of the equivalent program point, letting the target code re-execute side-effect free instructions as in Graal. `reconstruct` may also be used to rematerialize dead values in a VM, instead of keeping them alive in optimized code.

Acknowledgements. We are grateful to Gustavo Petri for suggesting the use of bisimulation in our theoretical framework, to Jan Vitek for some enlightening discussions, and to the anonymous PLDI reviewers and our shepherd Iulian Neamtiu for their many useful comments.

References

[1] Ali-Reza Adl-Tabatabai. 1996. *Source-Level Debugging of Globally Optimized Code.* Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA. Advisor(s) Gross, Thomas. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.73.5762. Accessed: 2016-06-18.

[2] Ali-Reza Adl-Tabatabai and Thomas Gross. 1996. Source-level Debugging of Scalar Optimized Code. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, New York, NY, USA, 33–43. https://doi.org/10.1145/231379.231388

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/349299.349303

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018).

[5] Earl T. Barr and Mark Marron. 2014. TARDIS: Affordable Time-travel Debugging in Managed Runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 67–82. https://doi.org/10.1145/2660193.2660209

[6] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *Proceedings of the 17th International Conference on Formal Methods (FM'11)*. Springer-Verlag, Berlin, Heidelberg, 200–214. http://dl.acm.org/citation.cfm?id=2021296.2021319

[7] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. 2010. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 708–725. https://doi.org/10.1145/1869459.1869517

[8] Abhilash Bhandari and V. Krishna Nandivada. 2015. Loop Tiling in the Presence of Exceptions. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 124–148. https://doi.org/10.4230/LIPIcs.ECOOP.2015.124

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (April 1986), 244–263. https://doi.org/10.1145/5397.5399

[10] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator Strength Reduction. *ACM Transactions on Programming Languages and Systems* 23, 5 (Sept. 2001), 603–625. https://doi.org/10.1145/504709.504710

[11] Max Copperman and Charles E. McDowell. 1993. A Further Note on Hennessy's "Symbolic Debugging of Optimized Code". *ACM Transactions Programming Languages and Systems* 15, 2 (April 1993), 357–365. https://doi.org/10.1145/169701.214526

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[13] Daniele Cono D'Elia and Camil Demetrescu. 2016. Flexible On-stack Replacement in LLVM. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*. ACM, New York, NY, USA, 250–260. https://doi.org/10.1145/2854038.2854061

[14] Thomas G. Evans and D. Lucille Darley. 1966. On-line Debugging Techniques: A Survey. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference (AFIPS '66 (Fall))*. ACM, New York, NY, USA, 37–50. https://doi.org/10.1145/1464291.1464295

[15] Stephen J Fink and Feng Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. IEEE Computer Society, 241–252. https://doi.org/10.1109/cgo.2003.1191549

[16] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2017. Correctness of Speculative Optimizations with Dynamic Deoptimization. *Proc. ACM Program. Lang.* 2, POPL, Article 49 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158137

[17] 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 *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

[18] Shu-yu Guo and Jens Palsberg. 2011. The Essence of Compiling with Traces. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 563–574. https://doi.org/10.1145/1926385.1926450

[19] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. 2012. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'12)*. Springer-Verlag, Berlin, Heidelberg, 278–293. https://doi.org/10.1007/978-3-642-27705-4_22

[20] John Hennessy. 1982. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems* 4, 3 (July 1982), 323–344. https://doi.org/10.1145/357172.357173

[21] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[22] Michael Hicks and Scott Nettles. 2005. Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.* 27, 6 (Nov. 2005), 1049–1096. https://doi.org/10.1145/1108970.1108971

[23] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[24] Urs Hölzle and David Ungar. 1994. A Third-generation SELF Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA '94)*. ACM, New York, NY, USA, 229–243. https://doi.org/10.1145/191080.191116

[25] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. 2000. FULLDOC: A Full Reporting Debugger for Optimized Code. In *Proceedings of the 7th International Symposium on Static Analysis (SAS '00)*. Springer, Berlin, Heidelberg, 240–259. https://doi.org/10.1007/978-3-540-45099-3_13

[26] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 327–337. https://doi.org/10.1145/1542476.1542513

[27] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. 2002. Proving Correctness of Compiler Optimizations by Temporal Logic. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/503272.503299

[28] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. 2004. Compiler Optimization Correctness by Temporal Logic. *Higher-Order and Symbolic Computation* 17, 3 (Sept. 2004), 173–206.

https://doi.org/10.1023/B:LISP.0000029444.99264.c0

[29] Nurudeen A. Lameed and Laurie J. Hendren. 2013. A Modular Approach to On-Stack Replacement in LLVM. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/2451512.2451541

[30] Sorin Lerner, Todd Millstein, and Craig Chambers. 2003. Automatically Proving the Correctness of Compiler Optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 220–231. https://doi.org/10.1145/781131.781156

[31] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. 2005. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 364–377. https://doi.org/10.1145/1040305.1040335

[32] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[33] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. 2012. Automating Object Transformations for Dynamic Software Updating. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 265–280. https://doi.org/10.1145/2384616.2384636

[34] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2008. THOR: A Tool for Reasoning About Shape and Arithmetic. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08)*. Springer-Verlag, Berlin, Heidelberg, 428–432. https://doi.org/10.1007/978-3-540-70545-1_41

[35] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

[36] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/93542.93578

[37] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA.

[38] Phoronix. 2016. Phoronix Test Suite (PTS). (2016). URL http://www.phoronix-test-suite.com/. Accessed: 2017-04-09.

[39] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag, London, UK, UK, 151–166. https://doi.org/10.1007/bfb0054170

[40] David Schneider and Carl Friedrich Bolz. 2012. The Efficient Handling of Guards in the Design of RPython's Tracing JIT. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '12)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/2414740.2414743

[41] Suriya Subramanian. 2010. *Dynamic Software Updates: A VM-Centric Approach*. Ph.D. Dissertation. The University of Texas at Austin, Austin, TX, USA. Advisor(s) McKinley, Kathryn. URL https://suriya.github.io/papers/suriya-thesis-final.pdf. Accessed: 2017-11-13.

[42] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. 2009. Dynamic Software Updates: A VM-centric Approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1542476.1542478

[43] Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2018. Hop, Skip, & Jump: Practical On-Stack Replacement for a Cross-Platform Language-Neutral VM. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/3186411.3186412

[44] Kunshan Wang, Yi Lin, Stephen M. Blackburn, Michael Norrish, and Antony L. Hosking. 2015. Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*. 321–336. https://doi.org/10.4230/LIPIcs.SNAPL.2015.321

[45] Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. 2017. One Compiler: Deoptimization to Optimized Code. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/3033019.3033025

[46] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[47] Le-Chun Wu, Rajiv Mirani, Harish Patil, Bruce Olsen, and Wen-mei W. Hwu. 1999. A New Framework for Debugging Globally Optimized Code. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 181–191. https://doi.org/10.1145/301618.301663

[48] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[49] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[50] Polle T. Zellweger. 1983. An Interactive High-level Debugger for Control-flow Optimized Programs. In *Proceedings of the Symposium on High-level Debugging (SIGSOFT '83)*. ACM, New York, NY, USA, 159–172. https://doi.org/10.1145/1006147.1006183

[51] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate Profiling in the Presence of Dynamic Compilation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 433–450. https://doi.org/10.1145/2814270.2814281