Formal Verification of SSA-Based Optimizations for LLVM

Jianzhou Zhao Santosh Nagarakatte[†] Milo M. K. Martin Steve Zdancewic

Computer and Information Science Department, University of Pennsylvania [†]Rutgers University

jianzhou@cis.upenn.edu santosh.nagarakatte@cs.rutgers.edu milom@cis.upenn.edu stevez@cis.upenn.edu

Abstract

Modern compilers, such as LLVM and GCC, use a *static single assignment* (SSA) intermediate representation (IR) to simplify and enable many advanced optimizations. However, formally verifying the correctness of SSA-based optimizations is challenging because SSA properties depend on a function's entire control-flow graph.

This paper addresses this challenge by developing a proof technique for proving SSA-based program invariants and compiler optimizations. We use this technique in the Coq proof assistant to create mechanized correctness proofs of several "micro" transformations that form the building blocks for larger SSA optimizations. To demonstrate the utility of this approach, we formally verify a variant of LLVM's `mem2reg` transformation in Vellvm, a Coq-based formal semantics of the LLVM IR. The extracted implementation generates code with performance comparable to that of LLVM's unverified implementation.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification - Correctness Proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs - Mechanical verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Operational semantics

General Terms Languages, Verification, Reliability

Keywords LLVM, Coq, single static assignment

1. Introduction

Compiler bugs can manifest as crashes during compilation, or, much worse, result in the silent generation of incorrect programs. Such mis-compilations can introduce subtle errors that are difficult to diagnose and generally puzzling to software developers. A recent study by Yang *et al.* [20] used random test-case generation to expose serious bugs in mainstream compilers including GCC, LLVM, and commercial compilers. Whereas few bugs were found in the front end of the compiler, various optimization phases of the compiler that aim to make generated programs faster were a prominent source of bugs.

Projects like CompCert [9, 16–18] are tackling the problem of compiler bugs by *mechanically verifying* the correctness of compilers. Indeed, although the aforementioned study uncovered many bugs in other compilers, the only bugs found in CompCert were in those parts of the compiler not (yet) formally verified. Yang *et al.* write [20]: *"The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users."*

Despite CompCert's groundbreaking compiler-verification efforts, there still remain many challenges in applying its technology to industrial-strength compilers. In particular, the original CompCert development, and the bulk of the subsequent work—with the notable exception of CompCertSSA [4]—did not use a *static single assignment (SSA)* [7] intermediate representation (IR) [9].

In SSA intermediate representations, each variable of a function is assigned statically only once, and each variable definition must dominate all of its uses in the control-flow graph. These SSA properties simplify or enable many compiler optimizations [14], including: constant folding, sparse conditional constant propagation, aggressive dead code elimination, global value numbering, global code motion, partial redundancy elimination, and inductive variable analysis. Consequently, open-source and commercial compilers such as GCC, LLVM, and Intel CC all use SSA-based intermediate representations when performing such optimizations.

Despite their importance, there are few mechanized formalizations of the correctness properties of SSA transformations. This paper tackles this problem by developing proof techniques suitable for mechanically verifying the correctness of SSA-based optimizations. We do so in the context of our Vellvm framework [21, 22], which formalizes the operational semantics of programs expressed in LLVM's SSA-based IR [10] and provides Coq [6] infrastructure to facilitate mechanized proofs of properties about transformations on the LLVM IR.

The key idea is to generalize the "SSA scoping" predicate, which is an invariant of the operational semantics, to more complex safety predicates. The main proof obligation then becomes a preservation argument, but one that takes the dominance relation of the SSA form into account. Instances of this idea are found in the literature (see, for example, Menon, *et al.* [13]), and related proof techniques have been recently used in CompCertSSA [4], which uses translation validation to check SSA optimizations (see Section 8). This work is the first to verify SSA algorithms in the context of an industrial-strength compiler like the LLVM.

To demonstrate the utility of these techniques, we use them to establish the correctness of a variant of `mem2reg`, a crucial LLVM pass that is responsible for promoting un-aliased local variables and stack-based temporary values into registers. We build our `vmem2reg` pass by "pipelining" several simpler transformations, such as load-after-store elimination, which are common building blocks of many SSA-based optimizations. To improve compilation time, we also investigate how to fuse such multiple such passes into one. To summarize our contributions, this paper:

- introduces general-purpose proof techniques for verifying SSA optimizations,
- shows that the proposed SSA proof techniques are viable for use in mechanized proofs, and verifies several simple transformations,

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06...$15.00

- describes and proves correct `vmem2reg`, an easier-to-verify variant of LLVM's `mem2reg` optimization, which is a key SSA-based transformation used by LLVM, and
- demonstrates that the extracted implementation of the fully verified `vmem2reg` is effective: The `vmem2reg` fully-verified pass yields an average speedup over a LLVM `-O0` of 77% versus a speedup of 81% speedup obtained by LLVM's implementation of `mem2reg`.

To streamline the explanation (Section 3), we describe the proof techniques in the context of Vminus, a simplified subset the full Vellvm framework [22], but one that still captures the essence of SSA. We mechanically verified *all* the claims of the paper both for Vminus and full Vellvm in Coq.[1]

2. Background

2.1 Program refinement

In this paper, we use program refinement to reason about the correctness of compilation. Following the CompCert project [9], we define program refinement in terms of programs' external behaviors (which include program traces of input-output events, whether a program terminates, and the returned value if a program terminates): a transformed program *refines* the original if the behaviors of the original program include all the behaviors of the transformed program. We define the operational semantics using traces of a labeled transition system.

Events	e	$::=$	$v = \mathit{fid}(\overline{v_j}^{\,j})$
Finite traces	t	$::=$	$\epsilon \mid e, t$
Finite or infinite traces	T	$::=$	$\epsilon \mid e, T$ (coinductive)

We denote one small-step of evaluation as $\mathit{config} \vdash S \xrightarrow{t} S'$: in program environment *config*, program state S transitions to the state S', recording events e of the transition in the trace t. An event e describes the inputs v_j and output v of an external function call named fid. $\mathit{config} \vdash S \xrightarrow{t}{}^{*} S'$ denotes the reflexive, transitive closure of the small-step evaluation with a finite trace t. $\mathit{config} \vdash S \xrightarrow{T} \infty$ denotes a diverging evaluation starting from S with a finite or infinite trace T. Program refinement is given by the following definition.

DEFINITION 1 (Program refinement).

1. $\mathbf{init}\,(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, S)$ *means S is the initial program state of the program prog with the main entry fid and inputs v_j.*
2. $\mathbf{final}\,(S, v)$ *means S is the final state with the return value v.*
3. $\Downarrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, t, v)$ *means* $\exists S\, S'.\ \mathbf{init}\,(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, S)$, $\mathit{config} \vdash S \xrightarrow{t}{}^{*} S'$ *and* $\mathbf{final}\,(S', v)$.
4. $\Uparrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, T)$ *means* $\exists S.\ \mathbf{init}\,(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, S)$ *and* $\mathit{config} \vdash S \xrightarrow{T} \infty$.
5. $\not\Downarrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, t)$ *means* $\exists S\, S'.\ \mathbf{init}\,(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, S)$, $\mathit{config} \vdash S \xrightarrow{t}{}^{*} S'$ *and S' is stuck.*
6. $\mathbf{defined}\,(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j})$ *means* $\forall\, t, \neg \not\Downarrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^{\,j}, t)$
7. prog_2 ***refines*** *program* prog_1, *written* $\mathit{prog}_1 \supseteq \mathit{prog}_2$, *if*
   - (a) $\mathbf{defined}\,(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^{\,j})$
   - (b) $\Downarrow(\mathit{prog}_2, \mathit{fid}, \overline{v_j}^{\,j}, t, v) \Rightarrow \Downarrow(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^{\,j}, t, v)$
   - (c) $\Uparrow(\mathit{prog}_2, \mathit{fid}, \overline{v_j}^{\,j}, T) \Rightarrow \Uparrow(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^{\,j}, T)$
   - (d) $\not\Downarrow(\mathit{prog}_2, \mathit{fid}, \overline{v_j}^{\,j}, t) \Rightarrow \not\Downarrow(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^{\,j}, t)$

Note that refinement requires only that a transformed program preserves the semantics of a well-defined original program, but does not constrain the transformation of undefined programs.

[1] Annotated Coq source available at `http://cis.upenn.edu/~stevez/vellvm/release.tgz`.

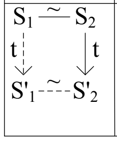
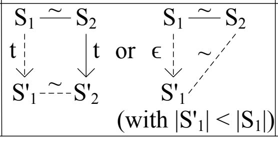
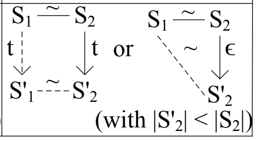
Lock-step	Right "option"	Left "option"
		

Figure 1. Backward simulation diagrams that imply program refinement. In each diagram, the program states of original and compiled programs are on the left and right respectively. A line denotes a relation $\sim$ between program states. Solid lines or arrows denote hypotheses; dashed lines or arrows denote conclusions.

We use the backward simulation diagrams in Figure 1 to prove that a program transformation satisfies the refinement property. The CompCert project uses similar diagrams for forward simulation [9]. At a high-level, we first need to find a relation $\sim$ between program states and their transformed counterparts. The relation must hold initially, imply equivalent returned values finally, and imply that stuck states are related. Then, depending on the transformation, we prove that a specific diagram holds: lock-step simulation is for variable substitution, right "option" simulation is for instruction removal, and left "option" simulation is for instruction insertion. Because the existence of a diagram implies that the source and target programs share traces, we can prove the equivalence of program traces by decomposing program transitions into matched diagrams. To ensure co-termination, the "option" simulations are parameterized by a measure of program states $|S|$ that must decrease to prevent "infinite stuttering" problems.

2.2 SSA

Static Single Assignment (SSA) form [7] is an intermediate representation distinguished by its treatment of temporary variables—each such variable may be defined only once, statically, and each use of the variable must be dominated by its definition with respect to the control-flow graph of the containing function.[2] Intuitively, the variable definition dominates a use if all possible execution paths to the use go through the definition first.

To maintain these invariants in the presence of branches and loops, SSA form uses ϕ-instructions, which act like control-flow dependent move operations. Such ϕ-instructions appear only at the start of a basic block and, crucially, they are handled specially in the dominance relation to "cut" apparently cyclic data dependencies.

The left part of Figure 2 shows an example program in SSA form, written using the stripped-down notation of Vminus (defined more formally in Section 3). The temporary r_3 at the beginning of the block labeled l_2 is defined by a ϕ-instruction: if control enters the block l_2 by jumping from basic block l_1, r_3 will get the value 0; if control enters from block l_2 (via the back edge of the branch at the end of the block), then r_3 will get the value of r_5.

The SSA form is good for implementing optimizations because it identifies variable names with the program points at which they are defined. Maintaining the SSA invariants thus makes definition and use information of each variable more explicit. Also, because each variable is defined only once, there is less mutable state to be considered (for purposes of aliasing, *etc.*) in SSA form, which makes certain code transformations easier to implement.

Program transformations like the one in Figure 2 are correct if the transformed program refines the original program (in the sense described above) and the result is well-formed SSA. Proving that

[2] In the literature, there are different variants of SSA forms [1]. We use LLVM' version: memory locations are not in SSA form; LLVM does not maintain any connection between a variable name in the IR and its original name in the source; the live ranges of variables can overlap.

Original	Transformed
$l_1 : \cdots$	$l_1 : \cdots$
$\quad \cdots$	$\quad r_4 := r_1 * r_2$
$\quad \mathbf{br}\, r_0\, l_2\, l_3$	$\quad \mathbf{br}\, r_0\, l_2\, l_3$
$l_2 : r_3 = \mathbf{phi}\, \mathbf{int} [0, l_1][r_5, l_2]$	$l_2 : r_3 = \mathbf{phi}\, \mathbf{int} [0, l_1][r_5, l_2]$
$\quad r_4 := r_1 * r_2$	
$\quad r_5 := r_3 + r_4$	$\quad r_5 := r_3 + r_4$
$\quad r_6 := r_5 \geq 100$	$\quad r_6 := r_5 \geq 100$
$\quad \mathbf{br}\, r_6\, l_2\, l_3$	$\quad \mathbf{br}\, r_6\, l_2\, l_3$
$l_3 : r_7 = \mathbf{phi}\, \mathbf{int} [0, l_1][r_5, l_2]$	$l_3 : r_7 = \mathbf{phi}\, \mathbf{int} [0, l_1][r_5, l_2]$
$\quad r_8 := r_1 * r_2$	
$\quad r_9 := r_8 + r_7$	$\quad r_9 := r_4 + r_7$

Figure 2. An SSA-based optimization. In the original program with entry l_1, $r_1 * r_2$ is a partial common expression for the definitions of r_4 and r_8, because there is no domination relation between r_4 and r_8. Therefore, eliminating the common expression directly is not correct: we cannot simply replace $r_8 := r_1 * r_2$ by $r_8 := r_4$ since r_4 is not available at the definition of r_8 if l_2 does not execute before l_3 runs. To transform this program, we might first move the instruction $r_4 := r_1 * r_2$ from the block l_2 to the block l_1, because the definitions of r_1 and r_2 must dominate the end of l_1, and l_1 dominates l_2. Then we can safely replace all the uses of r_8 by r_4, because the definition of r_4 in l_1 dominates l_3 and therefore dominates all the uses of r_8. Finally, r_8 is removed, because there are no uses of r_8.

such code transformations are correct is nontrivial because they involve non-local reasoning about the program.

3. Proof Techniques for SSA

This section describes the proof techniques we have developed for formalizing properties of SSA-style intermediate representations. To most clearly articulate the approach, we present the results using a language called Vminus, which is a minimalist SSA language containing only those features salient to the proof technique. Vminus is a subset of the LLVM IR formalized in Vellvm [22].

The key idea of the technique is to generalize the invariant used for Vminus's preservation lemma for proving safety to other predicates that are also shown to be invariants of the operational semantics. Crucially, these predicates all share the same form, which only constrains variable definitions that *strictly dominate* the current program counter.

The remainder of this section first presents the syntax and operational semantics of Vminus, then it gives the static semantics and proves safety (which in this context simply amounts to showing that all variables are well-scoped). With these properties established, we then show how to generalize the safety invariant to a form that is useful for proving program transformations correct and demonstrate its applicability to a number of standard optimizations. Because Vminus is such a stripped-down language, the relevant lemmas are relatively straightforward to establish; Section 4 shows how to scale the proof technique to the full Vellvm model of LLVM to verify the `mem2reg` pass.

3.1 The simple SSA language—Vminus

Syntax Figure 3 gives the syntax of Vminus. Every Vminus expression is of type integer. Operations in Vminus compute with values val, which are either temporaries r or constants $cnst$ that must be integer values. We use R to range over sets of identifiers.

All code in Vminus resides in a top-level function, whose body is composed of blocks b. Here, $\overline{b}$ denotes a list of blocks; we also use similar notation for other lists. As is standard, a basic block consists of a labeled entry point l, a series of ϕ nodes, a list of commands cs, and a terminator instruction tmn. In the following, we also use the label l of a block to denote the block itself.

Types	typ	$::=$	$\mathbf{int}$
Constants	$cnst$	$::=$	Int
Values	val	$::=$	$r \mid cnst$
Binops	bop	$::=$	$+ \mid * \mid \&\& \mid = \mid \geq \mid \leq \mid \cdots$
Right-hand-sides	rhs	$::=$	$val_1\ bop\ val_2$
Commands	c	$::=$	$r := rhs$
Terminators	tmn	$::=$	$\mathbf{br}\ val\ l_1\ l_2 \mid \mathbf{ret}\ typ\ val$
Phi Nodes	ϕ	$::=$	$r = \mathbf{phi}\ typ\ \overline{[val_j, l_j]}^{j}$
Instructions	$insn$	$::=$	$\phi \mid c \mid tmn$
Non-ϕs	ψ	$::=$	$c \mid tmn$
Blocks	b	$::=$	$l\ \overline{\phi}\ \overline{c}\ tmn$
Functions	f	$::=$	$\mathbf{fun}\ \{\overline{b}\}$

Values v	$::=$	Int	Locals δ	$::=$	$r \mapsto v$
Frames σ	$::=$	(pc, δ)	Prog Counters pc	$::=$	$l.i \mid l.\mathbf{t}$

$$\frac{[\![val]\!]_\delta = \lfloor v \rfloor \quad l_3 = (v ? l_1 : l_2) \qquad f[l_3] = \lfloor (l_3\, \overline{\phi}_3\, \overline{c}_3\, tmn_3) \rfloor \quad [\![\overline{\phi}_3]\!]^{l}_{\delta} = \lfloor \delta' \rfloor}{f \vdash (l, [], \mathbf{br}\ val\ l_1\ l_2, \delta) \longrightarrow (l_3, \overline{c}_3, tmn_3, \delta')} \ \text{E\_BR}$$

$$\frac{[\![val_1]\!]_\delta = \lfloor v_1 \rfloor \quad [\![val_2]\!]_\delta = \lfloor v_2 \rfloor \qquad c = r := val_1\ bop\ val_2 \quad \mathbf{eval}\,(bop, v_1, v_2) = v_3}{f \vdash (l, (c, \overline{c}), tmn, \delta) \longrightarrow (l, \overline{c}, tmn, \delta\{v_3/r\})} \ \text{E\_BOP}$$

Figure 3. Syntax and Operational Semantics of Vminus

The set of blocks making up the top-level function constitutes a control-flow graph with a well-defined entry point that cannot be reached from other blocks. We write $f[l] = \lfloor b \rfloor$ if there is a block b with label l in function f. Here, the $\lfloor\ \rfloor$ (pronounced "some") indicates that the function is partial (might return "none" instead).

As usual in SSA, the ϕ nodes join together values from a list of predecessor blocks of the control-flow graph—each ϕ node takes a list of (value, label) pairs that indicates the value chosen when control transfers from a predecessor block with the associated label. The commands c include the usual suite of binary arithmetic or comparison operations. We denote the right-hand-sides of commands by rhs. Block terminators ($\mathbf{br}$ and $\mathbf{ret}$) branch to another block or return a value from the function. We also use metavariable $insn$ to range over ϕ-nodes, commands and terminators, and non-phinodes ψ to represent commands and terminators.

Dynamic Semantics The operational semantics rules in Figure 3 are parameterized by the top-level function f, and relate evaluation frames σ before and after an evaluation step. An evaluation frame keeps track of the integer values v bound to local temporaries r in δ and current program counter. We also use $\sigma.pc$ and $\sigma.\delta$ to denote the program counter and locals of σ respectively. Because Vminus has no function calls, the rules ignore program traces. This simplification does not affect the essence of the proof techniques. Section 5 shows the full Vellvm semantics with traces.

Instruction positions are denoted by program counters pc: $l.i$ indicates the i-th command in the block l; $l.\mathbf{t}$ indicates the terminator of the block l. We write $f[pc] = \lfloor insn \rfloor$ if some $insn$ is at the program counter pc of function f. We also use $l.(i + 1)$ to denote the next program counter of $l.i$. When $l.i$ is the last command of block l, $l.(i + 1) = l.\mathbf{t}$. To simplify presentation of the operational semantics, we use $l, \overline{c}, tmn$ to "unpack" the instructions at a program counter in function f. Here, l is the current block, $\overline{c}$ and tmn are the instructions of l that are not executed yet. It is easy to see how the "block & offset" specification is equivalent to the "continuation commands" representation so we omit the details of the correspondence here.

$$\frac{\forall r.(\psi\,\mathbf{uses}\,r \Rightarrow r \in \mathbf{sdom}_f(pc))}{f \Vdash \psi \mathbin{@} pc}\ \text{NONPHI} \qquad \frac{f \rightsquigarrow l \Rightarrow (\overline{f, l \Vdash \phi_j}^{\,j} \wedge \overline{f \vdash c_i \mathbin{@} l.i}^{\,i} \wedge f \vdash tmn \mathbin{@} (l.\mathbf{t}))}{f \vdash l\,\overline{\phi_j}^{\,j}\,\overline{c_i}^{\,i}\,tmn}\ \text{WF\_B}$$

$$\frac{\mathbf{uniq}\,(\overline{l_j}^{\,j}) \quad \overline{l_j}^{\,j} = \mathbf{preds}\,(f, l) \quad \overline{\forall r_j.(val_j\,\mathbf{uses}\,r_j \Rightarrow r_j \in \mathbf{sdom}_f(l_j.\mathbf{t}))}^{\,j} \quad \mathbf{len}\,(\,\overline{[val_j, l_j]}^{\,j}\,) > 0 \quad \overline{f \vdash val_j : typ}^{\,j}}{f, l \Vdash r = \mathbf{phi}\,typ\,\overline{[val_j, l_j]}^{\,j}}\ \text{PHI}$$

$$\frac{f \vdash val_1 : \mathbf{int} \quad f \vdash val_2 : \mathbf{int}}{f \vdash r := val_1\ bop\ val_2}\ \text{WF\_BOP} \qquad \frac{f \vdash val : \mathbf{int} \quad f[l_1] = \lfloor b_1 \rfloor \quad f[l_2] = \lfloor b_2 \rfloor}{f \vdash \mathbf{br}\,val\,l_1\,l_2}\ \text{WF\_BR}$$

$$\frac{f \Vdash \psi \mathbin{@} pc \quad f \vdash \psi}{f \vdash \psi \mathbin{@} pc}\ \text{WF\_NONPHI} \qquad \frac{\mathbf{uniq}\,(\mathbf{defs}\,(f)) \quad \mathbf{uniq}\,(\mathbf{labels}\,(f)) \quad f = \mathbf{fun}\,\{\overline{b_j}^{\,j}\} \quad \overline{f \vdash b_j}^{\,j} \quad \mathbf{wf\_entry}\,f}{\vdash f}\ \text{WF\_F}$$

Figure 4. Static Semantics of Vminus (excerpt)

Most of the Vminus commands have straight-forward interpretation. The arithmetic and logic instructions are all unsurprising—the $[\![val]\!]_\delta$ function computes a value from the local state δ and val, looking up the meanings of variables in the local state as needed; **eval** implements arithmetic and logic operations. We use $[\![rhs]\!]_\delta$ to denote evaluating the right-hand-side *rhs* in the state δ.

There is one wrinkle in specifying the operational semantics when compared to a standard environment-passing call-by-value language. All of the ϕ instructions for a block must be executed atomically and with respect to the "old" local value mapping due to the possibility of self loops and dependencies among the ϕ nodes. For example the well-formed code fragment bellow has a circular dependency between r_1 and r_2

$$\begin{array}{ll} l_0: & \cdots \\ l_1: & r_1 = \mathbf{phi}\,\mathbf{int}[r_2, l_1][0, l_0] \\ & r_2 = \mathbf{phi}\,\mathbf{int}[r_1, l_1][1, l_0] \\ & r_3 := r_1 = r_2 \\ & \mathbf{br}\,r_3\,l_2\,l_1 \\ l_2: & \cdots \end{array}$$

If control enters this block from l_0, r_1 will map to 0 and r_2 to 1, which causes the conditional branch to fail, jumping back to the label l_1. The new values of r_1 and r_2 should be 1 and 0, and not 1 and 1 as might be computed if they were handled sequentially. This atomic update of the local state, similar to "parallel assignment," is handled by the $[\![\overline{\phi_3}]\!]^l_\delta$ function as shown in rule E_BR.

Static semantics Vminus requires that a program satisfy certain invariants to be considered well formed: every variable in the top-level function must dominate all its uses, and be assigned exactly, once statically. At a minimum, any reasonable Vminus transformation must preserve these invariants; together they imply that the program is in SSA form [7].

Figure 4 shows the judgments to check the SSA invariants with respect to the control-flow graph and program points of the function f. To explain the judgments we need the following definitions.

DEFINITION 2 (Domination).

1. *$val\,\mathbf{uses}\,r \triangleq val = r$.*
2. *$insn\,\mathbf{uses}\,r \triangleq \exists val.\ val\,\mathbf{uses}\,r \wedge val$ is an operand of $insn$.*
3. *A variable r is defined at a program counter pc of function f, written $f\,\mathbf{defines}\,r \mathbin{@} pc$ if and only if $f[pc] = \lfloor insn \rfloor$ and r is the left-hand side of $insn$. We write $\mathbf{defs}\,(f)$ to denote the set of all variables defined in f.*
4. *In function f, block l_1 **dominates** block l_2, written $f \vdash l_1 \succeq l_2$, if every path from the entry block of f to l_2 must go through l_1; l_1 **strictly dominates** l_2, written $f \vdash l_1 \succ l_2$, if every path from the entry block of f to l_2 must go through l_1 and $l_1 \neq l_2$.*
5. *In function f, pc_1 **strictly dominates** pc_2, written $f \vdash pc_1 \succ pc_2$, if pc_1 and pc_2 are at distinct blocks l_1 and l_2 respectively and $f \vdash l_1 \succ l_2$; if pc_1 and pc_2 are in the same block, then pc_1 appears earlier than pc_2.*
6. *$\mathbf{sdom}_f(pc)$ is the set of variables strictly dominating pc:*

$$\mathbf{sdom}_f(pc) = \{r \mid f\,\mathbf{defines}\,r \mathbin{@} pc' \text{ and } f \vdash pc' \succ pc\}$$

Rule WF_F of Figure 4 ensures that variables $\mathbf{defs}\,(f)$ defined in the top-level function are unique, which enforces the single-assignment part of the SSA property. Additionally, all block labels $\mathbf{labels}\,(f)$ in the function must be unique for a well-formed control-flow graph, and the entry block should have no predecessors ($\mathbf{wf\_entry}\,f$).

The rule WF_B checks that all instructions in reachable blocks (written $f \rightsquigarrow l$) satisfy the SSA domination invariant. Because unreachable blocks have no effects at runtime, the rule does not check them. Rule NONPHI ensures that a ψ at pc is strictly dominated by the definitions of all variables used by ψ. The rule PHI ensures that the number of incoming values is not zero, that all incoming labels are unique, and that the current block's predecessors is the same as the set of incoming labels. If an incoming value val_j from a predecessor block l_j uses a variable r_j at pc_j, then pc_j must strictly dominate the terminator of l_j. Importantly, this rule allows "cyclic" uses of SSA variables of the kind shown in the example above.

Dominance analysis Dominance analysis plays an important role in the type system, so we must first prove the following lemmas about the domination relations. They are needed to establish the SSA-based program properties in the following sections. Details about these lemmas can be found our prior work [21].

LEMMA 1 (Domination is transitive).

LEMMA 2 (Strict domination is acyclic).

By Lemma 1, $\mathbf{sdom}_f(pc)$ has the following properties:

LEMMA 3 (sdom step).

1. *If $l.i$ and $l.(i+1)$ are valid program counters of f, then $\mathbf{sdom}_f(l.(i{+}1)) = \mathbf{sdom}_f(l.i) \cup \{r\}$ where $f\,\mathbf{defines}\,r \mathbin{@} l.i$.*
2. *If $l.\mathbf{t}$ and $l'.0$ are valid program counters of f, and l' is a successor of l, then $\mathbf{sdom}_f(l'.0) - \mathbf{defs}\,(\overline{\phi}) \subseteq \mathbf{sdom}_f(l.\mathbf{t})$ where $\overline{\phi}$ are from the block l' and $\mathbf{defs}\,(\overline{\phi})$ denotes all variables defined by $\overline{\phi}$.*

3.2 Safety of Vminus

There are two ways that a Vminus program might get stuck. First, it might try to jump to an undefined label, but this property is ruled out statically by WF_BR. Second, it might try to access a variable whose value is not defined in δ. We can prove that this second case never happens by establishing the following safety theorem:

THEOREM 4 (Safety). *If $\vdash f$ and $f \vdash (entry.0, \emptyset) \longrightarrow^* \sigma$, then σ is not stuck.*

The proof takes the standard form using preservation and progress lemmas with the invariant for frames shown bellow:

$$\frac{pc \in f \quad \forall r.(r \in \mathbf{sdom}_f(pc) \Rightarrow \exists v.\delta[r] = \lfloor v \rfloor)}{f \vdash (pc, \delta)} \quad \text{WF\_FR}$$

This rule is similar to the predicate used in prior work for verifying the type safety of an SSA-based language [13]. The invariant WF_FR shows that a frame (pc, δ) is well-formed if every definition that strictly dominates pc is defined in δ. The initial program state satisfies this invariant trivially:

LEMMA 5 (Initial State). *If $\vdash f$ then $f \vdash (entry.0, \emptyset)$.*

The preservation and progress lemmas are straightforward—but note that they crucially rely on the interplay between the invariant on δ "projected" onto $\mathbf{sdom}_f(pc)$ (Lemma 3), and the PHI and NONPHI rules of the static semantics.

LEMMA 6 (Preservation). *If $\vdash f$, $f \vdash \sigma$ and $f \vdash \sigma \longrightarrow \sigma'$, then $f \vdash \sigma'$.*

LEMMA 7 (Progress). *If $\vdash f$, $f \vdash \sigma$, then σ is not stuck.*

3.3 Generalizing Safety to other SSA Invariants

The main feature of the preservation proof, Lemma 6, is that the constraint on $\mathbf{sdom}_f(pc)$ is an invariant of the operational semantics. But—*and this is a key observation*—we can parameterize rule WF_FR by a predicate P, which is an arbitrary proposition about functions and frames:

$$\boxed{\frac{\sigma.pc \in f \qquad P\, f\, (\sigma|_f)}{f, P \vdash \sigma} \quad \text{GWF\_FR}}$$

Here, $\sigma|_f$ is $(\sigma.pc, (\sigma.\delta)|_{(\mathbf{sdom}_f(\sigma.pc))})$ and we write $(\delta|_R)[r] = \lfloor v \rfloor$ iff $r \in R$ and $\delta[r] = \lfloor v \rfloor$ and observe that $\mathbf{dom}\,(\delta|_R) = R$. These restrictions say that we don't need to consider *all* variables: Intuitively, because SSA invariants are based on dominance properties, when reasoning about a program state we need only consider the variable definitions that strictly dominate the program counter in a given state.

For proving Theorem 4, we instantiated P to be:

$$P_{\text{safety}} \triangleq \lambda f.\, \lambda\sigma.\, \forall r.r \in \mathbf{dom}\,(\sigma.\delta) \Rightarrow \exists v.(\sigma.\delta)[r] = \lfloor v \rfloor$$

For safety, it is enough to show that each variable in the domination set is well defined at its use. To prove program transformations correct, we instantiate P with a different predicate, P_{sem}, that relates the syntactic definition of a variable with the semantic value:

$$\lambda f.\, \lambda\sigma.\, \forall r.f[r] = \lfloor rhs \rfloor \Rightarrow (\sigma.\delta)[r] \neq \cdot \Rightarrow (\sigma.\delta)[r] = [\![rhs]\!]_{(\sigma.\delta)}$$

Just as we proved preservation for P_{safety}, we can also prove preservation for P_{sem} (using Lemma 2):

THEOREM 8. *If $\vdash f$ and $f, P_{\text{sem}} \vdash \sigma$ and $f \vdash \sigma \longrightarrow \sigma'$, then $f, P_{\text{sem}} \vdash \sigma'$.*

As we show next, Theorem 8 can be used to justify the correctness of many SSA-based transformations. Instantiating P with other predicates can also be useful—Section 5 shows how.

3.4 The correctness of SSA-based transformations

Consider again the example code transformation from Figure 2. It, and many other SSA-based optimizations, can be defined by using a combination of simpler transformations: deleting an unused definition, substituting a constant expression for a variable, substituting one variable by another, or moving variable definitions. Each such transformation is subject to the SSA constraints—for example, we can't move a definition later than one of its uses—and each transformation preserves the SSA invariants. By pipelining these basic transformations, we can define more sophisticated SSA-based program transformations whose correctness is established by the composition of the proofs for the basic transformations.

In general, an SSA-based transformation from f to f' is *correct* if it preserves both well-formedness and program behavior.

1. Preserving well-formedness: if $\vdash f$, then $\vdash f'$.
2. Program refinement: if $\vdash f$, then $f \supseteq f'$ (see Section 2.1).

Each of the basic transformations mentioned above can be proved correct by using Theorem 8. For the sake of space, here we present only the correctness of variable substitution (though we proved correct all the mentioned transformations in our Coq development). Section 4 shows how to extend the transformations to implement memory-aware optimizations in the full Vellvm.

Variable substitution Consider the step of the program transformation from Figure 2 in which the use of r_8 on the last line is replaced by r_4 (this is valid only after hoisting the definition of r_4 so that it is in scope). This transformation is correct because both r_4 and r_8 denote the same value, and the definition of r_4 (after hoisting) strictly dominates the definition of r_8. In Figure 2, it is enough to do *redundant variable elimination*—this optimization lets us replace one variable by another when their definitions are syntactically equal; other optimizations, such as *global value numbering*, allow a coarser, more semantic, equality to be used. Proving them correct follows the same basic pattern as the proof shown below.

DEFINITION 3 (Redundant Variable). *In a function f, a variable r_2 is **redundant** with variable r_1 if:*

1. *f **defines** r_1 @ pc_1, f **defines** r_2 @ pc_2 and $f \vdash pc_1 \succ pc_2$*
2. *$f[pc_1] = \lfloor c_1 \rfloor$, $f[pc_1] = \lfloor c_2 \rfloor$ and c_1 and c_2 have syntactically equal right-hand-sides.*

We would like to prove that eliminating a redundant variable is correct, and therefore must relate a program f with $f\{r_1/r_2\}$. Since substitution does not change the control-flow graph, it preserves the domination relations.

LEMMA 9.

1. *$f \vdash l_1 \succcurlyeq l_2 \iff f\{r_2/r_1\} \vdash l_1 \succcurlyeq l_2$*
2. *$f \vdash pc_1 \succ pc_2 \iff f\{r_2/r_1\} \vdash pc_1 \succ pc_2$*

Applying Lemma 1 and Lemma 9, we have:

LEMMA 10. *Suppose that in f, r_1 is redundant with r_2. If $\vdash f$, then $\vdash f\{r_2/r_1\}$.*

Let two program states simulate each other if they have the same local state δ and program counter. We assume that the original program and its transformation have the same initial state.

LEMMA 11. *If $\vdash f$, r_2 is redundant with r_1 in f, and (pc, δ) is a reachable state, then*

1. *If val is an operand of a non-phinode at program counter pc, then $\exists v.\, [\![val]\!]_\delta = \lfloor v \rfloor \wedge [\![val\{r_1/r_2\}]\!]_\delta = \lfloor v \rfloor$.*
2. *If pc is $l_i.\mathbf{t}$, and l_i is a previous block of a block with ϕ-nodes $\overline{\phi_j}^{\,j}$, then $\exists\delta'.\, [\![\overline{\phi_j}^{\,j}]\!]_\delta^{l_i} = \lfloor \delta' \rfloor \wedge [\![\overline{\phi_j\{r_1/r_2\}}^{\,j}]\!]_\delta^{l_i} = \lfloor \delta' \rfloor$.*

Proof (sketch): The proof makes crucial use of Theorem 8. For example, to show part 1 for a source instruction $r := rhs$ (with transformed instruction $r := rhs\{r_1/r_2\}$) located at program counter pc, we reason like this: if r_2 is an operand used by rhs, then $r_2 \in \mathbf{sdom}_f(pc)$ and by Theorem 8, property P_{sem}, implies that $\delta[r_2] = [\![rhs_2]\!]_\delta$ for some rhs_2 defining r_2. Since r_1 is used as an operand in $rhs\{r_1/r_2\}$, similar reasoning shows that $\delta[r_1] =$

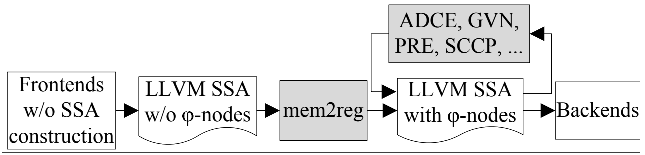


Figure 5. The tool chain of the LLVM compiler

$[\![rhs_1]\!]_\delta$, but since r_2 is redundant with r_1, we have $rhs_2 = rhs_1$, and the result follows immediately. □

Using Lemma 11, we can easily show the lock-step simulation lemma, which completes the correctness proof:

LEMMA 12. *If* $\vdash f$*,* r_2 *is redundant with* r_1 *in* f*,* $f\{r_1/r_2\} \vdash \sigma_1 \longrightarrow \sigma_2$*, then* $f \vdash \sigma_1 \longrightarrow \sigma_2$*.*

4. LLVM and `mem2reg`

From Vminus to Vellvm Vminus provides a convenient minimal setting in which to study SSA-based optimizations, but it omits many features necessary in a real intermediate representation. To demonstrate that our proof techniques can be used for practical compiler optimizations, we next show how to apply them to the LLVM IR used in Vellvm [22].

LLVM [10] (*Low-Level Virtual Machine*) is a robust, industrial-strength, and open-source compilation framework that competes with GCC in terms of compilation speed and performance of the generated code. LLVM uses a platform-independent SSA-based intermediate representation [10], and provides a large suite of optimization passes, including aggressive dead code elimination, global value numbering, partial redundancy elimination, and sparse conditional constant propagation, among others. Each transformation pass consumes and produces code in this SSA form, and they typically have the flavor of the code transformations described above in Section 3. Figure 5 depicts LLVM's tool chain.

The Vellvm infrastructure provides a Coq implementation of the full LLVM 3.0 intermediate language and defines (several) operational semantics along with some useful metatheory about the memory model. Figure 6 shows the additional Vellvm features needed to explain the following proofs; more details about the operational semantics can be found in our earlier Vellvm work [22].

A program *prog* includes a list of products *prod* that are either global constants or function definitions. The address of a global constant with type *typ* is a constant $typ * gid$ of type $typ*$. Vellvm also has memory operations that include stack allocation, loads, and stores. At runtime, a pointer in is represented by a block name *blk* and an offset *ofs* within the block. All globals are allocated before the start of a program and stored in a mapping g. A program state S is composed of a memory state M and a list of stack states $\overline{\sigma}$. We denote small-step evaluation by $config \vdash S \xrightarrow{t} S'$ where a program configuration *config* includes a program *prog* and globals. Judgment $\vdash prog$ denotes a well-formed program—it checks that all definitions of globals and functions are unique, each function is well-formed, *etc.*

4.1 The `mem2reg` Optimization Pass

A critical piece of LLVM's compilation strategy is the `mem2reg` pass, which takes code that is "trivially" in SSA form and converts it into a *minimal, pruned* SSA program [15]. This strategy simplifies LLVM's many front ends by moving work in to `mem2reg`. An SSA form is "minimal" if each ϕ is placed only at the dominance frontier of the definitions of the ϕ node's incoming variables [7]. A minimal SSA form is "pruned" if it contains only live ϕ nodes [15]. This pass enables many subsequent optimizations (and, in particular, backend optimizations such as register allocation) to work effectively.

Programs	$prog$	$::=$	$\overline{prod}$
Products	$prod$	$::=$	$gid = \mathbf{global}\ typ\ const \mid f$
Functions	f	$::=$	$\mathbf{define}\ typ\ fid(\overline{arg})\ \{\overline{b}\}$
Types	typ	$::=$	$\cdots \mid typ*$
Constants	$cnst$	$::=$	$\cdots \mid typ * gid$
Commands	c	$::=$	$\cdots$
		$\mid$	$r := \mathbf{alloca}\ typ$
		$\mid$	$r := \mathbf{load}\ (\ typ*\ )\ r_1$
		$\mid$	$\mathbf{store}\ typ\ val_1\ r_2$
		$\mid$	$option\ r = \mathbf{call}\ typ\ fid\ \overline{param}$
Values	v	$::=$	$\cdots \mid blk.ofs$
Globals	g	$::=$	$r \mapsto v$
Configurations	$config$	$::=$	$prog, g$
Allocas	α	$::=$	$\emptyset \mid blk, \alpha$
Frames	σ	$::=$	(f, pc, δ, α)
Call stacks	$\overline{\sigma}$	$::=$	$\emptyset \mid \sigma, \overline{\sigma}$
Program states	S	$::=$	$M, \overline{\sigma}$

Figure 6. The syntax and program states of Vellvm (excerpt)

Figure 7 demonstrates the importance of the `mem2reg` pass for LLVM's generated code performance. In our experiments, running *only* the `mem2reg` pass yields a 81% speedup (on average) compared to LLVM without any optimizations; doing the full suite of `-O1` level optimizations (which includes `mem2reg`) yields a speedup of 102%, which means that `mem2reg` alone captures all but %12 of the benefit of the `-O1` level optimizations. Comparison with `-O3` optimizations yields similar results. These observations make `mem2reg` an obvious target for our verification efforts.

The "trivial" SSA form is generated directly by compiler front ends, and it uses the **alloca** instruction to allocate stack space for *every* source-program local variable and temporary needed. In this form, an LLVM SSA variable is used either only locally to access those stack slots, in which case the variable is never live across two basic blocks, or it is a reference to the stack slot, whose lifetime corresponds to the source-level variable's scope. These constraints mean that no ϕ instructions are needed—it is extremely straightforward for a front end to generate code in this form.

As an example, consider this C program:

```
int i = 0; while (i<=100) i++; return i;
```

The "trivial" SSA form that might be produced by the frontend of a compiler is shown in the left-most column of Figure 8. The $r_0 := \mathbf{alloca}\,\mathbf{int}$ instruction on the first line allocates space for the source variable `i`, and r_0 is a reference from which local **load** and **store** instructions access `i`'s contents.

The `mem2reg` pass converts *promotable* uses of stack-allocated variables to SSA temporaries. An **alloca**'ed variable like r_0 is considered to be promotable, written $\mathbf{promotable}\,(f, r_0)$, if it is created in the entry block of function f and it doesn't escape—*i.e.,* its value is never written to memory or passed as an argument to a function call. The `mem2reg` pass identifies promotable stack allocations and then replaces them by temporary variables in SSA form. It does this by placing ϕ nodes, substituting each variable defined by a **load** with the previous value stored into the stack slot, and then eliminating the memory operations (which are now dead). The right-most column of Figure 8 shows the resulting pruned SSA program for this example.

Proving that `mem2reg` is correct is nontrivial because it makes significant, non-local changes to the use of memory locations and temporary variables. Furthermore, the specific `mem2reg` algorithm used by LLVM is not directly amenable to the proof techniques developed in Section 3—it was not designed with verification in mind, so it produces intermediate stages that break the SSA invariants or do not preserve semantics. This section therefore describes an alternate algorithm that is more suitable to formalization.

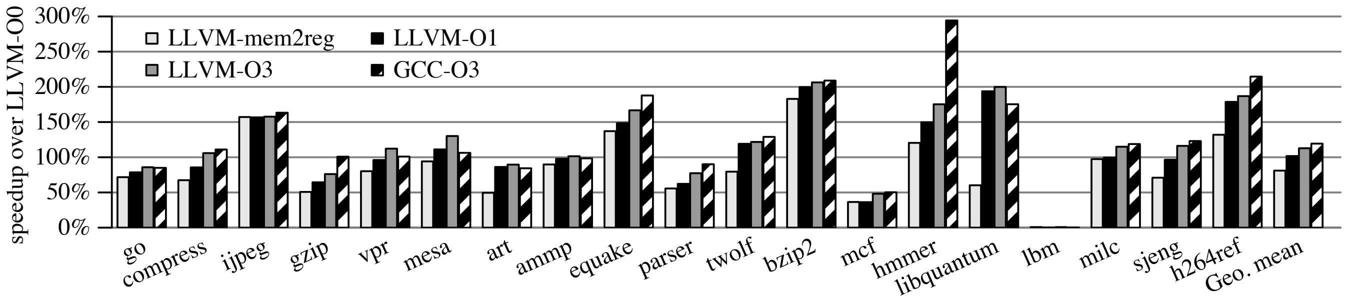


Figure 7. Normalized execution time improvement of the LLVM's mem2reg, LLVM's O1, and LLVM's O3 optimizations over the LLVM baseline with optimizations disabled. For comparison, GCC-O3's speedup over the same baseline is also shown.

"trivial" SSA	Maximal ϕ nodes placement	After LAS/LAA/SAS	After DSE/DAE	After ϕ nodes elimination
$l_1 : r_0 := \textbf{alloca int}$	$l_1 : r_0 := \textbf{alloca int}$	$l_1 : r_0 := \textbf{alloca int}$	$l_1 :$	$l_1 :$
$\textbf{store int}\ 0\ r_0$	$\textbf{store int}\ 0\ r_0$	$\textbf{store int}\ 0\ r_0$		
	$r_7 := \textbf{load}\,(\textbf{int}\,*)\,r_0$			
$\textbf{br}\ l_2$	$\textbf{br}\ l_2$	$\textbf{br}\ l_2$	$\textbf{br}\ l_2$	$\textbf{br}\ l_2$
$l_2 :$	$l_2 : r_6 = \textbf{phi}\,[r_7, l_1][r_9, l_3]$	$l_2 : r_6 = \textbf{phi}\,[0, l_1][r_9, l_3]$	$l_2 : r_6 = \textbf{phi}\,[0, l_1][r_4, l_3]$	$l_2 : r_6 = \textbf{phi}\,[0, l_1][r_4, l_3]$
	$\textbf{store int}\ r_6\ r_0$	$\textbf{store int}\ r_6\ r_0$		
$r_1 := \textbf{load}\,(\textbf{int}\,*)\,r_0$	$r_1 := \textbf{load}\,(\textbf{int}\,*)\,r_0$			
$r_2 := r_1 \leq 100$	$r_2 := r_1 \leq 100$	$r_2 := r_6 \leq 100$	$r_2 := r_6 \leq 100$	$r_2 := r_6 \leq 100$
	$r_8 := \textbf{load}\,(\textbf{int}\,*)\,r_0$			
$\textbf{br}\ r_2\ l_3\ l_4$	$\textbf{br}\ r_2\ l_3\ l_4$	$\textbf{br}\ r_2\ l_3\ l_4$	$\textbf{br}\ r_2\ l_3\ l_4$	$\textbf{br}\ r_2\ l_3\ l_4$
$l_3 :$	$l_3 : r_{10} = \textbf{phi}\,[r_8, l_2]$	$l_3 : r_{10} = \textbf{phi}\,[r_6, l_2]$	$l_3 : r_{10} = \textbf{phi}\,[r_6, l_2]$	$l_3 :$
	$\textbf{store int}\ r_{10}\ r_0$			
$r_3 := \textbf{load}\,(\textbf{int}\,*)\,r_0$	$r_3 := \textbf{load}\,(\textbf{int}\,*)\,r_0$			
$r_4 := r_3 + 1$	$r_4 := r_3 + 1$	$r_4 := r_{10} + 1$	$r_4 := r_{10} + 1$	$r_4 := r_6 + 1$
$\textbf{store int}\ r_4\ r_0$	$\textbf{store int}\ r_4\ r_0$	$\textbf{store int}\ r_4\ r_0$		
	$r_9 := \textbf{load}\,(\textbf{int}\,*)\,r_0$			
$\textbf{br}\ l_2$	$\textbf{br}\ l_2$	$\textbf{br}\ l_2$	$\textbf{br}\ l_2$	$\textbf{br}\ l_2$
$l_4 :$	$l_4 : r_{11} = \textbf{phi}\,[r_8, l_2]$	$l_4 : r_{11} = \textbf{phi}\,[r_6, l_2]$	$l_4 : r_{11} = \textbf{phi}\,[r_6, l_2]$	$l_4 :$
	$\textbf{store int}\ r_{11}\ r_0$	$\textbf{store int}\ r_{11}\ r_0$		
$r_5 := \textbf{load}\,(\textbf{int}\,*)\,r_0$	$r_5 := \textbf{load}\,(\textbf{int}\,*)\,r_0$			
$\textbf{ret int}\ r_5$	$\textbf{ret int}\ r_5$	$\textbf{ret int}\ r_{11}$	$\textbf{ret int}\ r_{11}$	$\textbf{ret int}\ r_6$

Figure 8. The SSA construction by the `vmem2reg` pass

4.2 The `vmem2reg` Algorithm

The `vmem2reg` algorithm is structured to lead to a clean formalism and yet still produce programs with effectiveness similar to the LLVM `mem2reg` pass. To demostrate the main ideas of `vmem2reg`, this section describes an algorithm that uses straightforward micro-pass pipelining. Section 6 presents a smarter way to "fuse" the micro passes, thereby reducing compilation time. Proving pipeline fusion correct is (by design) independent of the proofs for the `vmem2reg` algorithm shown in the section.

At a high level, `vmem2reg` (whose code is shown in Figure 10) traverses all functions of the program, applying the transformation `vmem2reg_fn` to each. Figure 9 depicts the main loop, which is an extension of Aycock and Horspool's SSA construction algorithm [3]. `vmem2reg_fn` first iteratively promotes each promotable **alloca** by adding ϕ nodes at the beginning of every block. After processing all promotable **alloca**s, `vmem2reg_fn` removes redundant ϕ nodes, and eventually will produce a program almost in pruned SSA form,[3] in a manner similar to previous algorithms [15].

[3] Technically, fully pruned SSA requires a more aggressive dead-ϕ-elimination pass that we omit for the sake of simplicity. Section 7 shows that this omission is negligible.

The transformation that `vmem2reg_fn` applies to each function is a composition of a series of micro transformations (LAS, LAA, SAS, DSE, and DAE, shown in Figure 9). Each of these transformations preserves the well-formedness and semantics of its input program; moreover, these transformations are relatively small and local, and can therefore be reasoned about more easily.

At each iteration of **alloca** promotion, `vmem2reg_fn` finds a promotable allocation r. Then ϕ-`nodes_placement` (code shown in Figure 10) adds ϕ nodes for r at the beginning of every block. To preserve both well-formedness and the original program's semantics, ϕ-`nodes_placement` also adds additional **load**s and **store**s around each inserted ϕ node. At the end of every block that has successors, ϕ-`nodes_placement` introduces a **load** from r, and stores the result in a fresh temporary; at the beginning of every block that has a predecessor, ϕ-`nodes_placement` first inserts a fresh ϕ node whose incoming value from a predecessor l is the value of the corresponding **load** added at the end of l, then inserts a **store** to r with the value of the new ϕ node.

The second column in Figure 8 shows the result of running the ϕ-node placement pass starting from the example program in its trivial SSA form. It is not difficult to check that this code is in SSA form. Moreover, the output program also preserves the meaning of the original program. For example, at the end of block l_1, the pro-

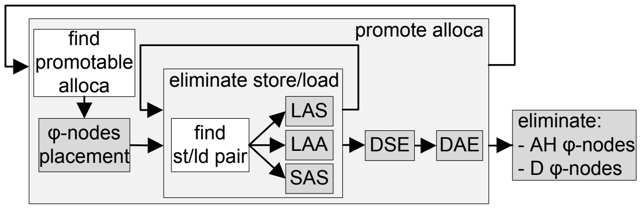


Figure 9. Basic structure of `vmem2reg_fn`

```
let vmem2reg $prog$ =
  map (function $f \rightarrow$ vmem2reg_fn $f$
                | $prod \rightarrow$ $prod$) $prog$
let $\phi$-nodes_placement $f$ $r$ =
  let $\mathbf{define}\, typ\, fid(\overline{arg})\, \{\overline{b}\}$ = $f$ in
  let (ldnms, phinms) = gen_fresh_names $\overline{b}$ in
  $\mathbf{define}\, typ\, fid(\overline{arg})\, \{$ (map
     (function $l\, \overline{\phi}\, \overline{c}\, tmn \rightarrow$
      let $r := \mathbf{alloca}\, typ \in f$ in
      let ($\overline{\phi}'$, $\overline{c}_1$) =
        match predecessors_of $f$ $l$ with
        | [] $\rightarrow$ ($\overline{\phi}$, $\overline{c}$)
        | $\overline{l_j}^{j}$ $\rightarrow$ let $\overline{r_j}^{j}$ = map (find ldnms) $\overline{l_j}^{j}$ in
                 let $r'$ = find phinms $l$ in
                 ($r' = \mathbf{phi}\, typ\, \overline{[r_j, l_j]}^{j} :: \overline{\phi}$, $\mathbf{store}\, typ\, r'\, r :: \overline{c}$)
        end in
      let $\overline{c}'$ = match successors_of $f$ $l$ with
                 | [] $\rightarrow \overline{c}_1$
                 | _ $\rightarrow$ let $r'$ = find ldnms $l$ in
                      $\overline{c}_1$ ++ [$r' := \mathbf{load}\, (\, typ * )\, r$]
                 end in
     $l\, \overline{\phi}'\, \overline{c}'\, tmn$) $\overline{b}$)}
let rec eliminate_stld $f$ $r$ =
  match find_stld_pair $f$ $r$ with
  | LAS ($pc_2$, $val_2$, $r_1$) $\rightarrow$
      eliminate_stld ($f\{val_2/r_1\} - r_1$) $r$
  | LAA $r_1$ $\rightarrow$ eliminate_stld ($f\{0/r_1\} - r_1$) $r$
  | SAS ($pc_1$, $pc_2$) $\rightarrow$ eliminate_stld ($f - pc_1$) $r$
  | NONE $\rightarrow f$
 end
```

Figure 10. The algorithm of `vmem2reg`

gram loads the value stored at r_0 into r_7. After jumping to block l_2, the value of r_7 is stored into the location r_0, which should contain the same values as r_7. Therefore, the additional store does not change the status of memory. Although the output program contains more temporaries than the original program, these temporaries are used only to connect inserted **load**s and **store**s, and so they do not interfere with the original temporaries.

To remove the additional **load**s and **store**s introduced by the ϕ-node placement pass and eventually promote **alloca**s to registers, `vmem2reg_fn` next applies a series of micro program transformations until no more optimizations can be applied.

First, `vmem2reg_fn` iteratively does the following transformations (implemented by `eliminate_stld` shown in Figure 10):

1. `LAS` (r_1, pc_2, val_2) "Load After Store": r_1 is **load**ed from r after a store of val_2 to r at program counter pc_2, and there are no other **store**s of r in any path (on the control-flow graph) from pc_2 to r_1. In this case, all uses of r_2 can be replaced by val_2, and the **load** can be removed.
2. `LAA` r_1 "Load After Alloca": As above, but the load is from an uninitialized memory location at r. r_1 can be replaced by LLVM's default memory value, and the **load** can be removed.
3. `SAS` (pc_1, pc_2): The **store** at program counter pc_2 is a store after the store at program counter pc_1. If both of them access r, and there is no **load** of r in any path (on the control-flow graph) from pc_1 to pc_2, then the **store** at pc_1 can be removed.

At each iteration step of `eliminate_stld`, the algorithm uses the function `find_stld_pair` to identify each of the above cases. Because the ϕ-node placement pass only adds a **store** and a **load** as the first and the last commands at each block respectively, `find_stld_pair` only needs to search for the above cases within blocks. This simplifies both the implementation and proofs. Moreover, `eliminate_stld` must terminate because each of its transformations removes one command. The third column in Figure 8 shows the code after `eliminate_stld`.

Next, the algorithm uses `DSE` (Dead Store Elimination) and `DAE` (Dead Alloca Elimination) to remove the remaining unnecessary **store**s and **alloca**s. The fourth column in Figure 8 shows the code after `DSE` and `DAE`.

Finally, `vmem2reg_fn` eliminates unnecessary and dead ϕ nodes. A ϕ-node is unnecessary [3] if ϕ is of the form $r = \mathbf{phi}\, typ\, \overline{[val_j, l_j]}^{j}$ where all the val_j's are either equal to r or to val. In this case, uses of r can be replaced by val, and the ϕ node can be removed. Aycock and Horspool [3] proved that when there is no such ϕ node in a reducible program, the program is of the minimal SSA form. The right-most column in Figure 8 shows the final output of the algorithm.

5. Correctness of `vmem2reg`

We prove the correctness of `vmem2reg` using the techniques developed in Section 3. At a high level, the correctness of `vmem2reg` is the composition of the correctness of each micro transformation of `vmem2reg` shown in Figure 10. Given a well-formed input program, each shaded box must produce a well-formed program that preserves the semantics of the input program. Moreover, the micro transformations except `DAE` and ϕ-nodes elimination must preserve the **promotable** predicate, because the correctness of subsequent transformations relies on fact that promotable allocations aren't aliased.

Formally, let $prog\{f'/f\}$ be the substitution of f by f' in $prog$, and let $(\!|f|\!)$ be a micro transformation of f applied by `vmem2reg`. $(\!|\_|\!)$ must satisfy:

1. Preserving **promotable**: when $(\!|\_|\!)$ is not `DAE` or ϕ-nodes elimination, if **promotable** (f, r), then **promotable** $((\!|f|\!), r)$.
2. Preserving well-formedness: if **promotable** (f, r) when $(\!|\_|\!)$ is ϕ-nodes placement, and $\vdash prog$, then $\vdash prog\{(\!|f|\!)/f\}$.
3. Program refinement: if **promotable** (f, r) when $(\!|\_|\!)$ is not ϕ-nodes elimination, and $\vdash prog$, then $prog \supseteq prog\{(\!|f|\!)/f\}$.

5.1 Preserving promotability

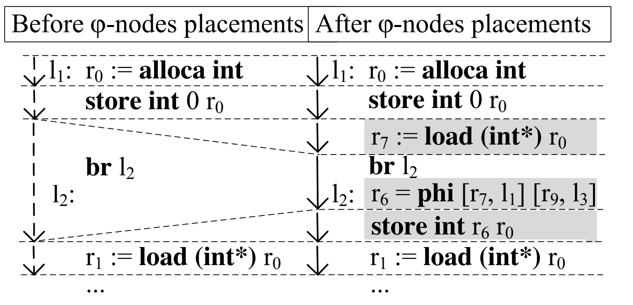
At the beginning of each iteration for promoting **alloca**s, the algorithm indeed finds promotable allocations.

LEMMA 13. *If prog* $\vdash f$*, and* `vmem2reg_fn` *finds a promotable allocation* r *in* f*, then* **promotable** (f, r).

We next show that ϕ-nodes placement preserves **promotable**:

LEMMA 14. *If* **promotable** (f, r),
then **promotable** $(\phi\text{–}nodes\_placement\ f\, r, r)$.

Each of the other micro transformations is composed of one or two more basic transformations: variable substitution, denoted by $f\{val/r\}$, and instruction removal, denoted by **filter check** f where **filter** removes an instruction $insn$ from f if **check** $insn =$



```
l1:  r0 := alloca int            l1:  r0 := alloca int
     store int 0 r0                   store int 0 r0
                                      r7 := load (int*) r0
     br l2                            br l2
l2:                              l2:  r6 = phi [r7, l1] [r9, l3]
                                      store int r6 r0
     r1 := load (int*) r0             r1 := load (int*) r0
     ...                              ...
```

Figure 11. The simulation relation for ϕ-node placement

false. For example, $f\{val_2/r_1\} - r_1$ (LAS) is a substitution followed by a removal in which $\mathbf{check}\, insn = \mathbf{false}$ iff $insn$ defines r_1; DSE of a promotable **alloca** r is a removal in which $\mathbf{check}\, insn = \mathbf{false}$ iff $insn$ is a store to r. We first establish that substitution and removal preserve $\mathbf{promotable}$.

LEMMA 15. *Suppose* $\mathbf{promotable}\,(f, r)$,

1. *If* $\neg(val_1\ \mathbf{uses}\, r)$, *then* $\mathbf{promotable}\,(f\{val_1/r_1\}, r)$.
2. *If* $\mathbf{check}\, insn = \mathbf{false} \Rightarrow$ *insn does not define r, then* $\mathbf{promotable}\,(\mathbf{filter}\,\mathbf{check}\, f, r)$.

We can show that the other micro transformations preserve $\mathbf{promotable}$ by checking the preconditions of Lemma 15.

LEMMA 16. *Suppose* $\mathbf{promotable}\,(f, r)$, *r is still* $\mathbf{promotable}$ *after* LAS, LAA, SAS *or* DSE.

The substituted value of LAS is written to memory by a **store** in f, which cannot use r because r is promotable in f. The substituted value of LAA is a constant that cannot use r trivially. Moreover, LAS, LAA, SAS and DSE remove only **load**s or **store**s.

5.2 Preserving well-formedness

It is sufficient to check the following conditions to show that a function-level transformation preserves well-formedness:

LEMMA 17. *Suppose*

1. $(\!|f|\!)$ *and* f *have the same signature.*
2. *if* $prog \vdash f$, *then* $prog\{(\!|f|\!)/f\} \vdash (\!|f|\!)$.

If $\vdash prog$, *then* $\vdash prog\{(\!|f|\!)/f\}$.

It is easy to see that all transformations vmem2reg applies satisfy the first condition. We first prove that ϕ-nodes placement preserves the second condition:

LEMMA 18. *If* $\mathbf{promotable}\,(f, r)$, $prog \vdash f$ *and let* f' *be* ϕ*–nodes_placement* $f\ r$, *then* $prog\{f'/f\} \vdash f'$.

Similarly, to reason about other transformations, we first establish that substitution and removal preserve well-formedness.

LEMMA 19. *Suppose* $prog \vdash f$,

1. *If* $f \vdash val_1 \succ r_2$, $f' = f\{val_1/r_2\}$, *then* $prog\{f'/f\} \vdash f'$.
2. *If* $\mathbf{check}\, insn = \mathbf{false} \Rightarrow f$ *does not use insn, and let* f' *be* $\mathbf{filter}\,\mathbf{check}\, f$, *then* $prog\{f'/f\} \vdash f'$.

Here, $f \vdash val_1 \succ r_2$ if $f \vdash r_1 \succ r_2$ when $val_1\ \mathbf{uses}\, r_1$. Note that the first part of Lemma 19 is an extension of Lemma 10 that only allows substitution on commands. In vmem2reg, LAS and ϕ-nodes elimination may transform ϕ-nodes.

LAS, LAA and ϕ-nodes elimination remove instructions after substitution. The following auxiliary lemma shows that the substituted definition is removable after substitution (by Lemma 2):

LEMMA 20. *If* $f \vdash val_1 \succ r_2$, *then* $f\{val_1/r_2\}$ *does not use* r_2.

By Lemma 19, Lemma 20 and the proofs [3], we have:

LEMMA 21. LAS, LAA, SAS, DSE, DAE *and* ϕ*-nodes elimination preserve well-formedness.*

5.3 Program refinement

The proofs of program refinement use the simulation diagrams in Section 2 and different instantiations of the GWF_FR rule we developed in Section 3, where instead of just a function f and frame σ, we now have a configuration *config* that also includes the program memory.

Promotability As we discussed above, the micro transformations (except ϕ-nodes elimination) rely on the $\mathbf{promotable}$ property. We start by establishing the invariants related to promotability, namely that promotable allocations aren't aliased. This proof is itself an application of GWF_FR.

The $\mathbf{promotable}$ property ensures that a promotable **alloca** of a function does not escape—the function can access the data stored at the allocation, but cannot pass the address of the allocation to other contexts. Therefore, in the program, the promotable **alloca** and all other pointers (in memory, local temporaries and temporaries on the stack) must not alias. Formally, given a promotable allocation r with type $typ*$ in f, we define $P_{\text{noalias}}(f, r, typ)$:

$$
\begin{array}{l}
\lambda config.\ \lambda S. \\
\quad \forall\, \overline{\sigma}_1 +\!\!+\, \sigma :: \overline{\sigma}_2 = S.\overline{\sigma}.\ f = \sigma.f \wedge [\![r]\!]_{\sigma.\delta} = \lfloor blk \rfloor \Longrightarrow \\
\qquad \exists v.\mathbf{load}\,(S.M, typ, blk) = \lfloor v \rfloor \\
\quad \wedge \quad \forall blk'.\forall typ'.\neg\mathbf{load}\,(S.M, typ', blk') = \lfloor blk \rfloor \\
\quad \wedge \quad \forall r' \neq r \Longrightarrow \neg[\![r']\!]_{\sigma.\delta} = \lfloor blk \rfloor \\
\quad \wedge \quad \forall \sigma' \in \overline{\sigma}_1.\forall r'.\neg[\![r']\!]_{\sigma'.\delta} = \lfloor blk \rfloor
\end{array}
$$

The last clause ensures that the **alloca** and the variables in the callees reachable from f do no alias. In CompCert, the translation from C#minor to Cminor uses properties (in non-SSA form) similar to $P_{\text{noalias}}(f, r, typ)$ to allocate local variables on stack.

LEMMA 22 (A promotable alloca is not aliased). *At any reachable program state* S, $config, P_{\text{noalias}}(f, r, typ) \vdash S$ *holds.*

The invariant holds initially. At all reachable states, the invariant holds because a promotable allocation cannot be copied to other temporaries, stored to memory, passed into a function, or returned. Therefore, in a well-defined program no external code can get its location by accessing other temporaries and memory locations. Importantly, the memory model ensures that from a consistent initial memory state, all memory blocks in temporaries and memory are allocated—it is impossible to forge a fresh pointer from an integer.

ϕ-node placement Figure 11 pictorially shows an example (which is the code fragment from Figure 8) of the simulation relation $\sim$ for proving that the ϕ-node placement preserves semantics. It follows left "option" simulation, because ϕ-node placement only inserts instructions. We use the number of unexecuted instructions in the current block as the measure function.

The dashed lines indicate where the two program counters must be synchronized. Although the pass defines new variables and **store**s (shaded in Figure 11), the variables are only passed to the new ϕ nodes, or stored into the promotable allocation; additional **store**s only update the promotable allocation with the same value. Therefore, by Lemma 22, $\sim$ requires that two programs have the same memory states and the original temporaries match.

LEMMA 23.
If $f' = \phi$*–nodes_placement* $f\ r$, *and* $\mathbf{promotable}\,(f, r)$, *and* $\vdash prog$, *then* $prog \supseteq prog\{f'/f\}$.

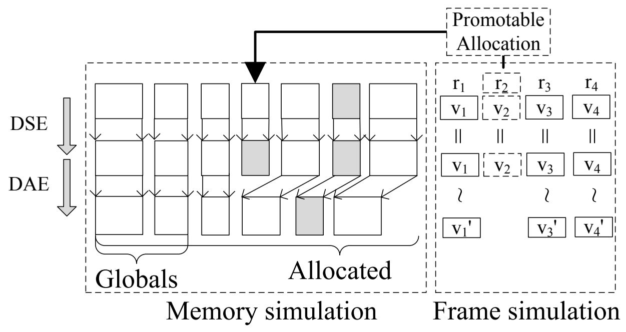


Figure 12. The simulation relation for `DSE` and `DAE`

The interesting case is to show that $\sim$ implies a correspondence between stuck states. Lemma 22 ensures that the promotable allocation cannot be dereferenced by operations on other pointers. Therefore, the inserted memory accesses are always safe.

LAS/LAA We present the proofs for the correctness of `LAS`. The proofs for the correctness of `LAA` is similar. In the code after ϕ-node placement of Figure 8, $r_7 := \mathbf{load}\,(\,\mathbf{int}\,*\,)\,r_0$ is an `LAS` of $\mathbf{store}\,\mathbf{int}\,0\,r_0$. We observe that at any program counter pc between the **store** and **load**, the value stored at r_0 must be 0 because $\mathbf{alive}\,(pc_1, pc_2)$ holds—the **store** defined at pc_1 is not overwritten by other writes until pc.

To formalize the observation, consider a promotable r with type $typ*$ in f. Suppose `find_stld_pair` f r = `LAS` (pc_2, val_2, r_1). Consider the invariant $P_{\text{las}}(f, r, typ, pc_2, val_2)$:

$$\begin{array}{l} \lambda config.\, \lambda S.\, \forall \sigma \in S.\overline{\sigma}. \\ \quad (f = \sigma.f \wedge [\![val_2]\!]_{\sigma.\delta} = \lfloor v_2 \rfloor \wedge [\![r]\!]_{\sigma.\delta} = \lfloor blk \rfloor \wedge \\ \quad \mathbf{alive}\,(pc_2, \sigma.pc)) \Longrightarrow \mathbf{load}\,(S.M, typ, blk) = \lfloor v_2 \rfloor \end{array}$$

Using Lemma 22, we have that:

LEMMA 24. *If* $\mathbf{promotable}\,(f, r)$*, then* $\mathbf{alive}\,(pc_2, r_1)$ *and at any reachable state S, config,* $P_{\text{las}}(f, r, typ, pc_2, val_2) \vdash S$ *holds.*

Let two programs relate to each other if they have the same program states. Lemma 24 establishes that the substitution in `LAS` is correct. The following lemma shows that removal of unused instructions preserves semantics in general.

LEMMA 25. *If* $\mathbf{check}\; insn = \mathbf{false} \Rightarrow f$ *does not use insn, and* $\vdash prog$*, then* $prog \supseteq prog\{\mathbf{filter\,check}\,f/f\}$.

Lemma 20 shows that the precondition of Lemma 25 holds after the substitution in `LAS`. Finally, we have that:

LEMMA 26. *`LAS` preserves semantics.*

SAS/DSE/DAE Here we discuss only the simulation relations used by the proofs. `SAS` removes a **store** to a promotable allocation overwritten by a following memory write. We consider a memory simulation that is the identity when the program counter is outside the `SAS` pair, but ignores the promotable **alloca** when the program counter is between the pair. Due to Lemma 22 and the fact that there is no **load** between a `SAS` pair, no temporaries or other memory locations can observe the value stored at the promotable **alloca** between the pair.

Figure 12 pictorially shows the simulation relations between the program states before and after `DSE` or `DAE`. Shaded memory blocks contain uninitialized values. The program states on the top are before `DSE`, where r_2 is a temporary that holds the promotable stack allocation and is not used by any loads. After `DSE`, the memory values for the promotable allocation may not match the original program's corresponding block. However, values in temporaries and all other memory locations must be unchanged (by Lemma 22). Note that unmatched memory states only occur after the promotable allocation; before the allocation, the two memory states should be the same.

The bottom part of Figure 12 illustrates the relations between programs before and after `DAE`. After `DAE`, the correspondence between memory blocks of the two programs is not bijective, due to the removal of the promotable **alloca**. However, there must exist a mapping $\sim$ from the output program's memory blocks to the original program's memory blocks. The simulation requires that all values stored in memory and temporaries (except the promotable allocation) are equal modulo the mapping $\sim$.

ϕ-nodes elimination Consider $r = \mathbf{phi}\,typ\,\overline{[val_j, l_j]}^{\,j}$ (an AH ϕ-node) where all the val_j's are either equal to r or some val'. Lemma 21 showed that $f \vdash val' \succ r$. Intuitively, at any pc that both val' and r strictly dominate, the values of val' and r must be the same. Consider the invariant $P_{\text{ah}}(f, r, val')$:

$$\begin{array}{l} \lambda config.\, \lambda S.\, \forall \sigma \in S.\overline{\sigma}. \\ \quad f = \sigma.f \wedge [\![r]\!]_{\sigma.\delta} = \lfloor v_1 \rfloor \wedge [\![val']\!]_{\sigma.\delta} = \lfloor v_2 \rfloor \Longrightarrow v_1 = v_2 \end{array}$$

LEMMA 27. *config,* $P_{\text{ah}}(f, r, val') \vdash S$ *holds for any reachable program state S.*

Lemma 27 establishes that the substitution in ϕ-nodes elimination is correct by using the identity relation. Lemma 20 and Lemma 25 show that removing dead ϕ-nodes is correct.

5.4 The correctness of `vmem2reg`

Our main result, fully verified in Coq, is the composition of the correctness proofs for all the micro program transformations:

THEOREM 28 (`vmem2reg` is correct). *If* $f' =$ *`vmem2reg`* f *and* $\vdash prog$*, then* $\vdash prog\{f'/f\}$ *and* $prog \supseteq prog\{f'/f\}$.

6. Pipeline Fusion

In `vmem2reg` (show in Figure 9), there are two loops of micro pass-pipelinings: `eliminate_stld` and the redundant ϕ-node elimination. This section gives a smarter pipelining `vmem2reg-O` that "fuses" the micro passes, reducing compilation time. For the sake of space, we present the fused version of `eliminate_stld`; the fused ϕ-node elimination follows similarly (see our Coq development). The design and correctness proofs described here should be applicable to other optimizations besides `vmem2reg`.

At a high level, Figure 13 gives `eliminate_stld_O`, the fused `eliminate_stld`, which first takes a pass to find all initial eliminations, fuses them, and then takes another pass that eliminates **load**'s and **store**'s in terms of fused eliminations.

We use actions ac to denote micro eliminations:

$$\text{Actions} \quad ac \quad ::= \quad r \mapsto val \;\mid\; pc \mapsto \#$$

Here, $r \mapsto val$ denotes `LAS` (r, pc, val) or `LAA` r with the default memory value val and $pc \mapsto \#$ denotes `SAS` (pc, pc'). We use AC to denote a list of actions, and $AC(f)$ to denote a single pass of f that, for each $r \mapsto val$, replaces all uses of r by val, then removes the definition of r and, for each $pc \mapsto \#$, removes the **store** at pc.

To find all initial actions, `eliminate_stld_O` traverses the blocks of a function, uses `find_stld_pairs_block` to find actions for each block, and then concatenates them. At each block, `find_stld_pairs_block` traverses each command (by `find_stld_pairs_cmd`), and uses `stld_state` to keep track of the search state: `ST_INIT` is the initial state; `ST_AL` typ records the element type of the promotable allocation; `ST_ST` pc val records the value stored by the latest **store** at pc to the promotable allocation.

```
let find_stld_pair_cmd $r$ acc $c$ =
  let (st, $AC$) = acc in
  match $c$ with
  | $r_0$ := alloca $typ$ →
      if $r$ = $r_0$ then (ST_AL $typ$, $AC$) else acc
  | $pc_0$ : store $typ$ $val_1$ $r_2$ →
      let st′ = ST_ST $pc_0$ $val_1$ in
      if $r$ = $r_2$ then
         match st with
         | ST_ST $pc$ _ → (st′, ($pc \mapsto \#, AC$))
         | _ →  (st′, $AC$)
         end
      else acc
  | $r_0$ := load ( $typ*$ ) $r_1$ →
      if $r$ = $r_1$ then
         match st with
         | ST_ST _ $val$ → (st, ($r_0 \mapsto val, AC$))
         | ST_AL $typ$ → (st, ($r_0 \mapsto$ undef $typ, AC$))
         | _ → acc
         end
      else acc
  | _ → acc
  end
let find_stld_pairs_block $r$ $b$ =
  let (_ _ $\overline{c}$ _) = $b$ in
  fold_left (find_stld_pair_cmd $r$) $\overline{c}$ (ST_INIT, ∅)
let rec fuse_actions $AC$ =
  match $AC$ with
  | ∅ → ∅
  | $r \mapsto val, AC'$ →
      let $AC''$ = fuse_actions $AC'$ in
      let $val'$ = find_parent $AC''$ $val$ in
      $r \mapsto val', AC''\{val'/r\}$
  | $pc \mapsto \#, AC'$ → $pc \mapsto \#$, fuse_actions $AC'$
  end
let eliminate_stld_O $r$ $f$ =
  let $fheader\{\overline{b}\}$ = $f$ in
  let $AC$ = flat_map (rev (snd
    (find_stld_pairs_block $r$))) $\overline{b}$ in
  (fuse_actions $AC$)($f$)
```

Figure 13. eliminate_stld of vmem2reg-O

eliminate_stld_O must fuse the initial actions before transforming f. Let values be vertices, and elements in actions be edges. A list of actions found from a well-formed function forms a forest because SSA ensures acyclicity of def/use chains. fuse_actions fuses the forest to be a forest with depth one: each non-root node in a tree of the original forest maps to the root of the tree. In fuse_actions, find_parent AC val returns the parent of val in the forest formed by AC if val has a parent; otherwise returns val; $AC\{val/r\}$ substitutes r in the codomain of AC by val. Suppose that eliminate_stld_O finds a list of actions: $r_4 \mapsto r_3, r_5 \mapsto r_4, r_2 \mapsto r_1, r_3 \mapsto r_2, r_6 \mapsto r_3, []$. fuse_actions returns $r_4 \mapsto r_1, r_5 \mapsto r_1, r_2 \mapsto r_1, r_3 \mapsto r_1, r_6 \mapsto r_1, []$.

The interesting part for the correctness of vmem2reg-O is showing that the fused pass produces the same output as the pipelined transformations:

LEMMA 29 (eliminate_stld_O is correct). *If prog* $\vdash f$*, then* *eliminate_stld* r f = *eliminate_stld_O* r f.

By Lemma 29 and Theorem 30, we have that

THEOREM 30 (vmem2reg-O is correct). *If* f' = *vmem2reg-O* f *and* $\vdash$ *prog, then* $\vdash$ *prog*$\{f'/f\}$ *and prog* $\supseteq$ *prog*$\{f'/f\}$.

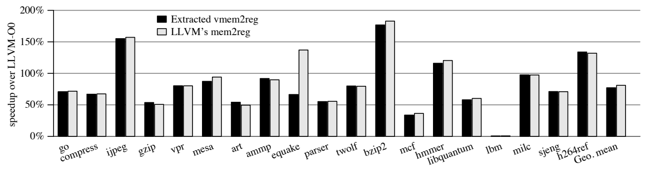


Figure 14. Execution speedup over LLVM -O0 for both the extracted vmem2reg and LLVM's original mem2reg pass.

7. Discussion and Evaluation

Coq Development Our Coq development for SSA optimizations consists of approximately 838 lines of algorithm implementations and 50k lines of (not particularly automated) correctness proof scripts and supporting infrastructure. We expect that much of these proofs can be reused for other SSA-based optimizations—that is the reason why we chose the pipeline of micro transformations structure. The development relies on about a dozen axioms, almost all of which define either the initial state of the machine (*i.e.,* where in memory functions and globals are stored) or the behavior of external function calls. One axiom asserts that memory alignment is a power of two, which is not necessary for LLVM programs in general, but is true of almost all real-world platforms.

Extracted vmem2reg and experimental methodology We used the Coq extraction mechanism to obtain a certified implementation of the vmem2reg optimization directly from the Coq sources. mem2reg is the first optimization pass applied by LLVM[4], so we tested the efficacy of the extracted implementation on LLVM IR bitcode generated directly from C source code using the clang compiler. At this stage, the LLVM bitcode is unoptimized and in "trivial" SSA form. To prevent the impact of this optimization pass from being masked by subsequent optimizations, we apply either LLVM's mem2reg or the extracted vmem2reg to the unoptimized LLVM bitcode and then immediately invoke the back-end code generator. We evaluate the performance of the resultant code on a 2.66 GHz Intel Core 2 processor running benchmarks selected from the SPEC CPU benchmark suite that consist of over 336k lines of C source code in total.

Figure 14 reports the execution time speedups (larger is better) over a LLVM's-O0 compilation baseline for various benchmarks. The left bar of each group shows the speedup of the extracted vmem2reg, which provides an average speedup of 77% over the baseline. The right bar of each group is the benefit provided by LLVM's mem2reg, which provides 81% on average; vmem2reg captures much of the benefit of the LLVM's mem2reg.

Comparing vmem2reg and mem2reg The vmem2reg pass differs from LLVM's mem2reg in a few ways. First, mem2reg promotes **alloca**s used by LLVM's intrinsics, while vmem2reg conservatively considers such **alloca**s to potentially escape, and so does not promote them. We determined that such intrinsics (used by LLVM to annotate the liveness of variable definitions) lead to almost all the difference in performance in the *equake* benchmark. Second, although vmem2reg deletes most unused ϕ-nodes, it does not aggressively remove them and, therefore, does not generate fully pruned SSA as mem2reg does. However, our results show that this does not impose a significant difference in performance.

Compilation time The focus of this work is *compiler correctness*, not *compilation time*. Although the code generated by vmem2reg is comparable to that of mem2reg, their compilation times differ, for two reasons. First, the extracted OCaml programs use purely functional data structures that impose $O(\log n)$ overhead compared to the efficient imperative hash tables available in C++. Second, the

[4] All results reported are for LLVM version 3.0.

pessimistic ϕ-node placement algorithm introduces unnecessary ϕ nodes. In our future work, we plan to use the minimal ϕ-node placement algorithm in `vmem2reg`, whose correctness is (by design) independent of the proofs presented in this paper. Moreover, unlike "all or nothing" translations, `vmem2reg` maintains the SSA invariants at each pass and can thus be applied incrementally to boost performance. In practice, should compilation time be an issue, it is always possible to use `mem2reg` for development and `vmem2reg` to ensure correctness for a final release.

8. Related Work and Conclusions

Verifying the correctness of compiler transformations is an active research area with a sizable amount of literature. We focus here on the work relevant to SSA-based optimizations.

CompCertSSA CompCertSSA [4] improves the CompCert compiler with a verified SSA-based middle-end and a GVN optimization pass. CompCertSSA verified a *translation validator* for an SSA construction algorithm that takes imperative variables to variables in a pruned SSA form. In contrast, our work fully verifies the SSA construction pass `vmem2reg` for LLVM directly. A bug in the CompCertSSA compiler will cause the validator to abort the compilation, whereas verifying the compiler rules out such a possibility. More pragmatically, translation validation is harder to apply in the context of LLVM, because the compiler infrastructure was not created with verification in mind. For example, the CompCertSSA translations maintain a close mapping between source and target variable names so that simulation can be checked by simple erasure; this is not feasible in the LLVM framework. The CompCertSSA project reports performance measurements of only small benchmarks totaling about 6k lines, whereas we have tested our pass on 336k lines, including larger programs.

Unsurprisingly, the CompCertSSA and Vellvm proofs share some similarities. For example, CompCertSSA's GVN proof uses an invariant similar to the one in our Theorem 8 and Lemma 12. However, the LLVM's strategy of promoting **alloca**s means that our proofs need a combination of both SSA and aliasing properties to prove correctness. Moreover, our proof technique of pipelining "micro" transformations is novel, and should be broadly applicable.

Other related work In less closely related work, Mansky *et al.* designed an Isabelle/HOL framework that uses control-flow graph rewrites to transform programs and uses temporal logic and model-checking to specify and prove the correctness of program transformations [11]. They verified an SSA construction algorithm in the framework. Other researchers have formalized specific SSA-based optimizations by using SSA forms with different styles of semantics: an informal semantics that describes the intuitive idea of the SSA form [7]; an operational semantics based on a matrix representation of ϕ nodes [19]; a data-flow semantics based term graphs using the Isabelle/HOL proof assistant [5]. Matsuno *et al.* defined a type system equivalent to the SSA form and proved that dead code elimination and common subexpression elimination preserve types [12]. There are also conversions between the programs in SSA form and functional programs [2, 8].

Conclusion We have presented a proof technique for formally verifying SSA-based compiler optimizations. Using the Coq proof assistant, we fully mechanized the proof technique and the correctness of several micro optimizations. For the full LLVM IR semantics in Vellvm, we have formalized and implemented an extractable SSA optimization pass `vmem2reg` that is an easier-to-prove variant of LLVM's `mem2reg` pass but that delivers most of its benefits.

Acknowledgments We would like to thank Delphine Demange, Vivien Durey, and Dmitri Garbuzov for their comments and suggestions about this work.

References

[1] Static Single Assignment Book, 2012. Working draft available at http://ssabook.gforge.inria.fr/latest/book.pdf.

[2] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4): 17–20, April 1998. ISSN 0362-1340.

[3] J. Aycock and N. Horspool. Simple generation of static single assignment form. In *CC*, 2000.

[4] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - Static Single Assignment meets CompCert. In *ESOP*, 2012.

[5] J. O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing code generation from SSA form: A comparison between two formal correctness proofs in Isabelle/HOL. *Electron. Notes Theor. Comput. Sci.*, 141(2):33–51, 2005.

[6] *The Coq Proof Assistant Reference Manual (Version 8.3pl1)*. The Coq Development Team, 2011.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13:451–490, 1991.

[8] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *IR*, number 3, 1995.

[9] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, December 2009. ISSN 0168-7433.

[10] *The LLVM Reference Manual (Version 3.0)*. The LLVM Development Team, 2011. http://llvm.org/releases/3.0/docs/LangRef.html.

[11] W. Mansky and E. L. Gunter. A framework for formal verification of compiler optimizations. In *ITP*, 2010.

[12] Y. Matsuno and A. Ohori. A type system equivalent to static single assignment. In *PPDP*, 2006.

[13] V. S. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A. Adl-Tabatabai, and L. Petersen. A verifiable SSA program representation for aggressive compiler optimization. In *POPL*, 2006.

[14] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[15] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *POPL*, 1995.

[16] J.-B. Tristan and X. Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *POPL*, 2008.

[17] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *PLDI*, 2009.

[18] J. B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.

[19] B. Yakobowski. Étude sémantique d'un langage intermédiaire de type Static Single Assignment. Rapport de dea (Master's thesis), ENS Cachan and INRIA Rocquencourt, Sept. 2004.

[20] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[21] J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *CPP*, 2012.

[22] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. This research was funded in part by DARPA contract HR0011-10-9-0008 and ONR award N000141110596. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1116682, CCF-1065166, and CCF-0810947. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.