A Translation Validation Algorithm for LLVM Register Allocators

Zhengyao Lin
University of Illinois at Urbana-Champaign
USA
zl38@illinois.edu

Theodoros Kasampalis
University of Illinois at Urbana-Champaign
USA
kasampa2@illinois.edu

Vikram Adve
University of Illinois at Urbana-Champaign
USA
vadve@illinois.edu

***Abstract*—Register allocation is a crucial and complex phase of any modern production compiler. In this work, we present a translation validation algorithm that verifies each single instance of register allocation. Our algorithm is external to the compiler and independent of the register allocation algorithm. We support all major register allocation optimizations such as live range splitting, register coalescing, and rematerialization. We developed a prototype of this approach for the production-quality register allocation phase of LLVM. We evaluated this prototype for compiling the source code of SPECint 2006 benchmark, and we were able to verify the register allocation of over 88% of supported functions in the benchmark, using all 4 register allocators available in LLVM 5.0.2.**

I. Introduction

Register allocation is a phase present in every modern optimizing compiler to determine the mapping from program variables to physical registers of the target instruction set architecture (ISA). In general, register allocation is NP-complete by a reduction from graph coloring [6], and modern compilers use sub-optimal heuristics together with a complex set of transformations that aim to give a best-effort solution: spilling [5], register coalescing [11], live range splitting [8], and rematerialization [4].

Since register allocation is often an essential and complex phase of the compilation pipeline, it is paramount to ensure its correctness. An ideal approach would be to develop the register allocator along with a formal correctness proof. However, due to the complexity of register allocation algorithms, this is not even done for the state-of-the-art verified C compiler, CompCert [21]. A far more serious limitation is that such *verified compilation* approaches are usually impractical to apply to existing, widely used production compilers such as GCC and LLVM, due to their large code base that is not designed with verification in mind.

In this paper, we study an alternative approach, *translation validation* (TV), to ensure the correctness of register allocation. Instead of proving the correctness of a compiler uniformly for all input, translation validation aims at proving the correctness of a single instance of compilation, by showing that the generated output program after register allocation refines (or is equivalent to) the input program. Such a proof can be done with few or no changes to the compiler, thus providing a practical verification solution for existing compilers.

Previous work on TV mostly focuses on validating the end-to-end compilation pipeline, including many optimization and code-generation passes (sometimes hundreds, as in GCC) [9], [12], [26], [32]. These systems are too complex for the purpose of validating individual phases, generally more computationally expensive, and also are not modular, so they are unable to isolate failures to individual compiler passes, which can be crucial during compiler development. See Section V for more details on related work.

In this work, we present a TV system that can prove the correctness of register allocation in a full-fledged production compiler system, such as LLVM. The key to our system is an inference algorithm based on dataflow analysis that generates *verification conditions* (VC) which imply the correctness of register allocation. Such VC is then verified by a previous developed proof checker KEQ [19] to ensure the soundness.

Our inference algorithm has three important features that make it practical and effective for use in production compilers. First, it is independent of the register allocation algorithm and can infer the correspondence between virtual registers in the input code and physical registers and/or spill memory locations in the output code solely by analyzing the input and output programs. Second, it does not require additional assistance from the compiler itself, making it especially easy to retrofit into existing compilers. Third, the algorithm is able to handle important and widely-used optimizations in register allocation, such as live range splitting, register coalescing, and certain types of rematerialization, as well as local optimizations done to reduce register pressure, like rescheduling and commuting of operands. Thus, we arrive at a highly practical compilation verification solution for the complex register allocation phase of a production compiler.

We have implemented our inference algorithm for register allocation in LLVM, and used it with the KEQ tool and the semantics definition of intermediate x86 programs, as described in [19]. We evaluated our TV System on 11 benchmark programs in SPECint 2006 [34]. The register allocation of more than 88% of all supported functions was successfully verified, using all four register allocators in LLVM 5.0.2. The remaining failures are discussed in Section IV-A; they are primarily due to inadequacies of our inference algorithm to handle specific edge cases as well as performance reasons

related with symbolic execution. So these failures can be fixed by mostly engineering effort. We also reintroduced two actual register allocation bugs to LLVM and verified that our system does not validate miscompilations resulting from these bugs, as discussed in Section IV-B.

To summarize, our main *contributions* in this paper are:

- A black-box algorithm (Section III) that infers verification conditions for the correctness of register allocation. The algorithm supports major optimizations such as live range splitting and register coalescing and does not require SMT solvers or other computationally expensive techniques. Combined with the previous work of KEQ and their x86 semantics, we derive a sound translation validator for register allocators in LLVM.
- Experimental results (Section IV) showing that Our TV system is able to validate the compilation of more than 88% of supported functions in eleven programs from the SPECint 2006 benchmark suite.

Furthermore, the KEQ algorithm and x86 semantics were previously used to prove the correctness of the Instruction Selection (ISel) phase of the LLVM system. Our work demonstrates that we can reuse these two components *unchanged* for register allocation as well, thus showing that it is possible and practical to reuse a significant part of a TV system [19] for a completely different compilation pass. This reusability is important in the long-term to enable modular translation validation of complex production compilers.

We organize the rest of the paper as follows. We briefly introduce KEQ and its formalism for proving program equivalence in Section II. We describe our inference algorithm for VC generation, which is our main technical contribution, in Section III. We show the evaluation of our tool on the SPECint 2006 test suite and on the reproduced register allocation bugs in Section IV. Finally, we discuss related work in Section V and conclude the paper in Section VI.

II. PRELIMINARIES

A. KEQ: A Transformation-Independent Equivalence Checker

In this work, we take advantage of the modular TV system presented in [19]. Specifically, the system is based on KEQ, a *language- and transformation-independent* equivalence checker. KEQ is implemented as a tool within the $\mathbb{K}$ framework [29], which is a language framework for defining operational semantics of programming languages. These semantic definitions are used to automatically derive tools for the defined language, and KEQ also follows this methodology.

KEQ accepts the input and output language semantics (written in $\mathbb{K}$), an input program, an output program (potentially after some transformations), and a set of verification conditions (VCs). The VCs are in the form of *synchronization points*: pairs of symbolic states for the input/output programs that are known to be related, along with a set of constraints on the variables in the two programs. We will refer to these constraints as *invariants*.

An example input/output program pair before and after register allocation is shown in Figure 1, with the locations of synchronization points marked (p_1 through p_4). The invariants for these synchronization points are shown in Figure 3. Each synchronization point symbolically describes pairs of states of program (a) and (b). For example, the synchronization point p_2 is placed at the edge from the basic block `.main` to `.loop`, and the invariant says that p_2 describes pairs of concrete states of program (a) and (b) at p_2 satisfying $\texttt{\%vr5} = \textbf{eax'} \wedge \texttt{\%vr4} = \textbf{edi'}$.

KEQ would then check whether, starting from any synchronization point (i.e. any pair of concrete states satisfying the invariant), the two programs would always reach another synchronization point in finite steps. So by a coinductive proof, one can conclude that starting from any pair of concrete states satisfying p_1, either both programs do not terminate, or they reach p_4. Finally, the invariants at p_1 and p_4 ensure exactly that the two programs have the same return value **eax** if they have the same input argument **edi**.

KEQ formalizes this idea using *cut-simulation*, proposed by Kasampalis *et al.* [19], which is a weaker variant of simulation [31] to capture more transformations. We review some definitions and results in the following. For simplicity, we only informally define these notions and we refer readers to [19, Section 7] for more formal definitions.

Definition 1 (Cut): Given a program P (as a transition system), a set of states $\mathcal{C}$ is a *cut* if for any complete trace τ from the starting state of P,

- If τ is finite, then τ starts and ends in $\mathcal{C}$.
- If τ is infinite, then τ visits $\mathcal{C}$ infinitely often.

Definition 2 (Cut-simulation): Given two programs P and P' and let $\mathcal{C}$ and $\mathcal{C}'$ be cuts for their transition systems, a relation $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{C}'$ between the cut states of P and cut states of P' is a *cut-simulation* iff for any $(s, s') \in \mathcal{R}$ such that s reaches some state $t \in \mathcal{C}$ through non-cut states, s' also reaches some state $t' \in \mathcal{C}'$ through non-cut states and $(t, t') \in \mathcal{R}$.

The above definitions are given in a language-independent manner. When we specialize KEQ to a specific pair of languages and programs, we would first define a binary relation $\mathcal{A}$ called the *acceptability relation* that restricts the pairs of states allowed to appear in any proposed cut-simulation. For instance, in the case of x86 programs, we relate in $\mathcal{A}$ all pairs of states that are not entry or exit points of the programs, and only relate entry points that have the same input and memory (such as p_1), and output points with the same memory and return value (such as p_4).

Then as a direct consequence of [19, Theorem 7.9], if we can find a cut-simulation between two programs contained in such $\mathcal{A}$, we have the following notion of program equivalence:

Theorem 1 (Soundness): Given two x86 programs P and P', if there exists a cut-simulation $\mathcal{R} \subseteq \mathcal{A}$ between the transition systems induced by P and P', then for any two complete traces τ and τ' of P, P', if τ and τ' starts in $\mathcal{A}$ (i.e. they begin with the same input arguments, memory, stack, etc.):

- If τ terminates in a state s, then τ' terminates in a state s', in which case s and s' have the same memory and return value.

```
.main:
   ; p1
   1 COPY %vr4, edi
   2 movl %vr5, 0

.loop:
   ; p2 p3
   3 PHI %vr0, (%vr4, .main), (%vr3, .loop)
   4 PHI %vr1, (%vr5, .main), (%vr2, .loop)
   5 addl %vr2, %vr1, %vr0
   6 addl %vr3, %vr0, -1
   7 jne .loop

.end:
   8 COPY eax, %vr2
   ; p4
   9 ret
```

(a) *Before Register Allocation*

```
.main:
   ; p1
   1 movl eax 0

.loop:
   ; p2 p3

   2 addl eax, eax, edi
   3 addl edi, edi, -1
   4 jne .loop

.end:

   ; p4
   5 ret
```

(b) *After Register Allocation*

Fig. 1: A simple program that computes the sum of $1 + \cdots +$ **edi** in Virtual x86 before and after Register Allocation (with the greedy allocator of LLVM under `-O2`). Comments in red show the program points for which synchronization points are placed by our inference algorithm.

Sync Point	*Prev BB*	*Invariant*
p_1	-	$\top$
p_2	`.main`	`%vr4` = **edi'** $\wedge$ `%vr5` = **eax'**
p_3	`.loop`	`%vr3` = **edi'** $\wedge$ `%vr2` = **eax'**
p_4	*any*	**eax** = **eax'**

Fig. 2: Invariants after the first iteration.

Sync Point	*Prev BB*	*Invariant*
p_1	-	**edi** = **edi'**
p_2	`.main`	`%vr4` = **edi'** $\wedge$ `%vr5` = **eax'**
p_3	`.loop`	`%vr3` = **edi'** $\wedge$ `%vr2` = **eax'**
p_4	*any*	**eax** = **eax'**

Fig. 3: Invariants at synchronization points for the programs in Figure 1, where **eax'** and **edi'** refer to the registers in the second program.

- If τ is infinite, then τ' is infinite, and they synchronize in $\mathcal{A}$ infinitely often.

Moreover, the intuitive verification algorithm described in the example above is generalized in [19] and proven sound. This reduces the work to finding a cut-simulation $\mathcal{R}$ that satisfies our acceptability $\mathcal{A}$. In our implementation, the inference algorithm proposes a set of synchronization points that symbolically describe $\mathcal{R}$. If KEQ can verify that $\mathcal{R}$ is indeed a cut-simulation, then the above notion of equivalence holds.

B. Virtual x86

In LLVM, the input and output programs of register allocators are represented in Machine IR [23]. Machine IR is a generic low level representation that can be parameterized by the specific opcodes of a target instruction set architecture and retains some higher-level features such as an infinite number of (virtual) registers, SSA form, function signatures, etc. We use *Virtual x86* to refer to the Machine IR specialized to the x86-64 instruction set [16]. We use the syntax illustrated by the example in Figure 1 to represent Virtual x86 code.

III. Synchronization Point Inference Algorithm for Register Allocation

We use KEQ and the Virtual x86 semantics definition from [19] (summarized in Section II), both unmodified, to develop our TV prototype for the register allocation phase of the LLVM compiler [22]. The additional component required to arrive at a fully functional TV system is an inference algorithm for the synchronization points that witness the correctness of register allocation.

Our inference algorithm is a backward dataflow analysis and operates on the input and output programs without any assistance from the compiler. The main observation is that an optimizing register allocator only uses a limited portion of the full semantics of the input language, in particular copy and phi instructions, uses and definitions of other instructions, etc., but not necessarily the full semantics of every instruction. So in our inference algorithm, we only capture such necessary fragment, resulting in a more tractable and maintainable solution. Even though such underapproximation by itself is unsound, the use of KEQ ensures that no true-negatives occur.

We start with an overview of the algorithm in Section III-A, then give a concrete example in Section III-B. We fill in the details of the algorithm in Section III-C and Section III-D. We discuss common register allocation optimizations we support in Section III-E and some limitations in Section III-F.

A. Overview of the Inference Algorithm

In this section, we give an overview of our algorithm for inferring synchronization points.

We will use P and P' to denote the input and output functions. We assume that P and P' have isomorphic CFGs; This is true in the LLVM register allocation phase, and is likely to be general enough for many register allocators in modern optimizing compilers. For simplicity of presentation we also assume that P and P' have disjoint register names. For example, we use names such as **eax** to denote physical registers in P and names such as **eax'** to denote physical registers in P'.

The algorithm consists of the following steps:

1) We promote any spilled stack frames in Virtual x86 to virtual registers.
2) We place synchronization points at the following program points:
   - The entry and exit points of both functions.
   - Every incoming edge of a non-entry/non-exit block.
   - Before and after each call to an external function.
3) We combine the two CFGs into a more abstract *product CFG* with nodes being the synchronization points and an edge between any synchronization points s_1 and s_2 iff there are program paths from s_1 to s_2 in both P, P'. This is similar to the notion of product CFGs/programs defined in previous work such as [7], [12].
4) We perform a backward dataflow analysis on the product CFG until it reaches a solution that assigns an invariant to each synchronization point.

In a common form of a backward dataflow analysis [3], our algorithm uses the following semi-lattice and transfer function:

- The semi-lattice for the dataflow values is the set $\mathit{Invariant}$ of invariants modulo equivalence, with the semi-lattice operation being formula conjunction, infimum being $\top$, and supremum being $\bot$. The syntax of invariants is shown in Figure 4.
- The transfer function $\mathit{Transfer}_{s_1,s_2} : \mathit{Invariant} \to \mathit{Invariant}$ on each edge (s_1, s_2) is described in Section III-C.
- The initial value at each synchronization point is $\top$.

Intuitively, $\top$ in our case means that the state at a particular point is unconstrained by any invariant; while if a $\bot$ invariant appears in any reachable point then it means that our heuristics (as explained in Section III-D) is too strict and have not considered all possible states.

B. Example for Inferring Invariants

Recall the example in Figure 1 which is a simple arithmetic program that computes the sum $1 + \cdots +$ **edi**. Functions (a) and (b) are respectively the input and output of register allocation in LLVM. Function (a) is still in SSA form and physical registers are only used for parameter passing (line 1) and return values (line 9). The two functions also use some of the so-called pseudo-instructions in Virtual x86, namely **COPY** (which copies between registers) and **PHI** (which denotes a phi node). The output function (b) uses only physical registers and the SSA form has been destructed.

The algorithm would first place synchronization points p_1 through p_4 as described in the overview in Section III-A. Then it would construct the product CFG as shown in Figure 5.

The transfer function for each edge (p_i, p_j) (which is detailed in Section III-C) works by first computing a Gen invariant (independent of the input invariant) by "matching" the pair of program paths connecting p_i and p_j. For example, consider the edge (p_2, p_3) associated with the program paths of lines 3-7 in function (a) and lines 2-4 in function (b). From these two lists of instructions, we would match the uses of line 5 in (a) with line 2 in (b), which gives us an inferred constraint

$$\varphi \coloneqq \texttt{\%vr0} = \textbf{edi'} \wedge \texttt{\%vr1} = \textbf{eax'}$$

This constraint is then transferred from the points at line 5 in (a) and line 2 in (b) backwards. When the transfer function encounters the **PHI** instruction at line 4, it would substitute `%vr1` with its definition `%vr5` (because the predecessor of p_2 is `.main`). Similarly, when the transfer function encounters **PHI** at line 3, it would replace `%vr0` with `%vr4`. As a result, the Gen invariant associated with (p_2, p_3) would be

$$\begin{aligned}&\varphi[\texttt{\%vr4}/\texttt{\%vr0}][\texttt{\%vr5}/\texttt{\%vr1}]\\ \equiv\ &\texttt{\%vr4} = \textbf{edi'} \wedge \texttt{\%vr5} = \textbf{eax'}\end{aligned}$$

Then for every input invariant, the transfer function performs the same operation backwards and takes the conjunction of the result with the Gen invariant.

Therefore, after the first iteration (recall that the initial invariant at each synchronization point is $\top$), the invariant at each synchronization point is shown in Figure 2.

After the second iteration, the invariants at p_2, p_3, p_4 have no changes. For p_1, the invariant $\texttt{\%vr4} = \textbf{edi'} \wedge \texttt{\%vr5} = \textbf{eax'}$ at p_2 is propagated back. When the transfer function encounters the copy instructions (**COPY** and **movl**), it replaces the the defined register with the source operand. So the resulting new invariant at p_1 is $\textbf{edi} = \textbf{edi'} \wedge 0 = 0$ which is simplified to $\textbf{edi} = \textbf{edi'}$.

Hence, the invariants converge after the second iteration to the final invariants in Figure 3, which concludes our example.

C. Transfer Function

For any edge (s_1, s_2) in the product CFG, the transfer function $\mathit{Transfer}_{s_1,s_2}$ has three components:

- Computing the Gen invariant by matching the instructions in the pair of paths from s_1 to s_2. This is elaborated in Section III-D
- Removing equalities that involves registers defined/killed by *non-copy* instructions using the Kill function defined in Figure 7.
- Propagating the resulting invariant through a pair of program paths using the $\mathit{Propagate}$ function defined in Figure 6.

Since we placed synchronization points at each basic block, we can assume that for each edge (s_1, s_2) in the product CFG, there are two unique program paths in P, P' from s_1 to s_2.

$$n \in \mathbb{N} \quad m \in \{1, \ldots, 64\} \quad r \text{ is a register in } P \text{ and } P'$$

$$t, t' \in \mathit{Term} ::= n \mid r$$
$$\varphi, \varphi' \in \mathit{Invariant} ::= \top \mid \bot \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid t = t' \pmod{2^m}$$

Fig. 4: Syntax of invariants. We may omit $\pmod{2^m}$ if the bit width is irrelevant or clear.

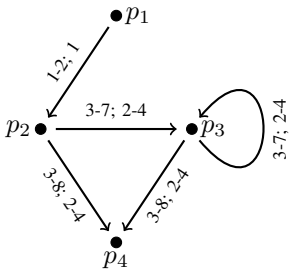

Fig. 5: The product CFG for the example in Figure 1. The edge labels represent the line numbers for the program paths connecting synchronization points in both functions (i.e. 1-2; 1 means that the program path in (a) connecting p_1 and p_2 is line 1-2, and that in (b) is line 1).

So let $\mathcal{I}$ be the list of instructions from s_1 to s_2 in P and let $\mathcal{I}'$ be the that in P'. $\mathit{Transfer}_{s_1,s_2}$ is defined as

$$\mathit{Transfer}_{s_1,s_2}(\mathcal{I}, \mathcal{I}', \varphi) \coloneqq$$
$$\mathit{Gen}(\mathcal{I}, \mathcal{I}') \wedge \mathit{Propagate}(s_1, \mathcal{I} \cdot \mathcal{I}', \mathit{Kill}(\mathcal{I}, \mathcal{I}', \varphi))$$

where $\cdot$ denotes list concatenation and $\mathit{Propagate}$ function is extended from the singleton version in Figure 6 inductively by $\mathit{Propagate}(s, I \cdot \mathcal{I}, \varphi) \coloneqq \mathit{Propagate}(s, I, \mathit{Propagate}(s, \mathcal{I}, \varphi))$ and $\mathit{Propagate}(s, (), \varphi) \coloneqq \varphi$. The Gen function will be defined in Section III-D.

Intuitively, $\mathit{Propagate}$ function captures the semantics of copy and phi instructions while Kill and Gen underapproximate the semantics of other instructions.

By definition, $\mathit{Propagate}$ (which only does substitution) and Kill functions are distributive, while Gen is independent of φ. So $\mathit{Transfer}_{s_1,s_2}$ is distributive. Therefore, the dataflow analysis can be implemented using the usual fixpoint iteration algorithm and it would terminate [17].

D. Gen function

As aforementioned, Gen function is defined by "matching" the instructions in $\mathcal{I}$ and $\mathcal{I}'$. This serves as heuristics to underapproximate the semantics of non-copy and non-phi instructions.

More specifically, we say that some $I_i \in \mathcal{I}$ *matches* some $I'_j \in \mathcal{I}'$ if they have the same opcode and same number of uses. Suppose they do match, let their uses be $r_1, \ldots, r_l$ and $r'_1, \ldots, r'_l$, respectively, and let

$$\phi_{I_i,I'_j} \coloneqq r_1 = r'_1 \pmod{2^{k_1}} \wedge \cdots \wedge r_l = r'_l \pmod{2^{k_l}}$$

where $k_1, \ldots, k_l \in \{1, \ldots, 64\}$ are bit widths of the matches that can be inferred from the opcode or register names (if not we default it to 64). Then we use $\mathit{Propagate}$ to transfer ϕ_{I_i,I'_j} backwards to the synchronization point s_1:

$$\mathit{Gen}_{I_i,I'_j} \coloneqq \mathit{Propagate}(s, (I_1, \ldots, I_{i-1}, I'_1, \ldots, I'_{j-1}), \phi_{I_i,I'_j})$$

In the example in Section III-B, we match two **`addl`** instructions in Figure 1 at line 5 in (a) and line 2 in (b). Their uses are `%vr1`, `%vr0` and **`eax'`**, **`edi'`** respectively, so the local constraint in this case is

$$\phi_{\text{line 5(a),line 2(b)}} \equiv \texttt{\%vr0} = \texttt{edi'} \pmod{2^{32}}$$
$$\wedge \texttt{\%vr1} = \texttt{eax'} \pmod{2^{32}}$$

where the bit width 32 is inferred from the opcode **`addl`**. Then their Gen constraint is computed by propagating through the two **`PHI`** instructions:

$$\begin{aligned} & \mathit{Gen}_{\text{line 5(a),line 2(b)}} \\ \equiv\ & \mathit{Propagate}(p_2, (\text{line 3(a)}, \text{line 4(a)}), \phi_{\text{line 5(a),line 2(b)}}) \\ \equiv\ & \texttt{\%vr4} = \texttt{edi'} \pmod{2^{32}} \wedge \texttt{\%vr5} = \texttt{eax'} \pmod{2^{32}} \end{aligned}$$

Finally, to compute $\mathit{Gen}(\mathcal{I}, \mathcal{I}')$, we find any matching M between instructions in $\mathcal{I}$ and $\mathcal{I}'$ in the sense defined above, and take

$$\mathit{Gen}(\mathcal{I}, \mathcal{I}') \coloneqq \bigwedge_{(I,I') \in M} \mathit{Gen}_{I,I'}$$

In our implementation, as another heuristic, we take a maximal matching between the two lists of instructions.

The Gen constraint defined above is a sufficient condition for the results of two matched instructions to be equal. This heuristic can be improved if we consider more properties of the actual instructions being matched.

For example, **`addl`** in x86 is a two-address instruction where the first register is both a source and destination register. To reduce the register pressure and make the first register reusable, a register allocator may permute the operands of **`addl`** if the second operand is not live after the add instruction. To capture such commutativity, we can take the constraint to be a disjunction for two possibilities. Suppose we are matching **`addl`** r_1, r_2 and **`addl`** r'_1, r'_2, we can take

$$\phi \coloneqq (r_1 = r'_1 \wedge r_2 = r'_2) \vee (r_1 = r'_2 \wedge r_2 = r'_1)$$

In our implementation, we use the constraint above for commutative operations such as add and xor.

Another example is that we can match instructions with different opcodes but the same semantics. For instance, the

$$Propagate(s, I, \varphi) \coloneqq \begin{cases} \varphi[t/r_{dst}] & I \text{ has the semantics of copying a register/constant } t \text{ to } r_{dst} \\ \varphi[r_{src}/r_{dst}] & I \equiv \texttt{PHI } r_{dst} \ \ldots (r_{src}, Pred(s)) \ldots \\ \varphi & \text{Otherwise} \end{cases}$$

Fig. 6: The *Propagate* function, which takes input a synchronization point s, an instruction I, and an invariant φ. Here *Pred* (s) is the predecessor basic block of the synchronization point s. $\varphi[t/x]$ is the result of substituting x in φ with a term t.

$$Kill(\mathcal{I}, \mathcal{I}', t = t') \coloneqq \begin{cases} \top & t, t' \text{ are killed by some } \textit{non-copy/non-phi} \text{ instructions in } \mathcal{I} \text{ or } \mathcal{I}' \text{ after propagation} \\ \bot & \text{One of } t \text{ and } t' \text{ is killed in } \mathcal{I} \text{ or } \mathcal{I}' \text{ but not the other} \\ t = t' & \text{Otherwise} \end{cases}$$

$$Kill(\mathcal{I}, \mathcal{I}', \bot) \coloneqq \bot$$

$$Kill(\mathcal{I}, \mathcal{I}', \top) \coloneqq \top$$

$$Kill(\mathcal{I}, \mathcal{I}', \varphi \vee \varphi') \coloneqq Kill(\mathcal{I}, \mathcal{I}', \varphi) \vee Kill(\mathcal{I}, \mathcal{I}', \varphi')$$

$$Kill(\mathcal{I}, \mathcal{I}', \varphi \wedge \varphi') \coloneqq Kill(\mathcal{I}, \mathcal{I}', \varphi) \wedge Kill(\mathcal{I}, \mathcal{I}', \varphi')$$

Fig. 7: The inductively-defined *Kill* function that underapproximates the semantics of non-copy instructions. $\mathcal{I}$ and $\mathcal{I}'$ denote lists of instructions. By t, t' being killed after propagation we mean that if some register r is defined in a copy instruction I by another register r', which is killed by a non-copy instruction before I, then we would still consider r being killed.

two-address **addl** instruction is sometimes translated to an equivalent version using **leal** in order to reduce register pressure. Therefore, we also consider such special cases in our implementation.

E. Optimizations in Register Allocation

Register allocators commonly use optimizations such as live range splitting [8], register coalescing [11], and rematerialization [4]. We discuss in this section whether our inference algorithm supports these optimizations.

1) Live Range Splitting and Register Coalescing: Live range splitting inserts a copy instruction within the live range of a register to split it into two distinct registers. As a result, interference between registers is reduced. Register coalescing, on the other hand, merges the source and destination registers of a copy instruction to simplify the interference graph. These optimizations will not affect our inference algorithm since only copy instructions are involved.

2) Rematerialization: Rematerialization recomputes the value of a register before its use, in the case when spilling is more costly than such recomputation.

A common case in the register allocators of LLVM is when a constant initialization of register is moved closer to the use of the register. Our inference algorithm supports this case since such initialization essentially has the semantics of copying a constant into a register and our invariant syntax includes constants as terms.

However, our inference algorithm does not identify which instructions are the results of rematerialization, thus not supporting this optimization in general.

F. Limitations

Our inference algorithm has two major limitations, which we leave as future work to solve.

1) Restricted to register allocation: Our current formulation only captures the semantics of copy and phi instructions, so it is limited to register allocation. Furthermore, in our algorithm, we make the assumption that the CFGs of two functions are isomorphic. This allows us to simply related basic block and construct a product CFG based on that. This strong assumption will likely not work for other global program transformations, which may techniques such as in [12] to search for a viable product CFG.

2) The Gen function requires engineering effort: In Section III-D, we discussed a few examples of how the matching heuristics of the *Gen* function can be more close to the actual semantics of the instructions. We have not developed a general approach to do this automatically, so such heuristics have to be engineered manually.

IV. Evaluation of the TV system for LLVM's register allocation

We evaluate the Translation Validation prototype in two ways. First, we apply the prototype to a substantial subset of the source code of SPECint 2006 benchmark [34] to evaluate its usefulness in a complex, real-world use case. Some of the test cases (like the widely used GCC compiler) are especially important for ensuring software reliability. Second, we reintroduced two register allocator bugs originally found in LLVM 3.5 and LLVM 7.0 in order to verify whether our prototype rejects translations affected by those bugs, and preferably also produces enough information for a compiler developer to diagnose the locations and potential causes of such failures.

A. Application on the SPECint 2006 Benchmark

In order to evaluate the Translation Validation prototype for a complex and important application, we applied it to a large

subset of the source code of the SPECint 2006 benchmark. We tested on 11 benchmark programs written in C and C++. For each program, we compiled the source code into LLVM IR using `clang-5.0.2` at optimization level -O0, then translated to Virtual x86 using the default instruction selection (ISel) pass of LLVM 5.0.2. We are using this setting because it has previously been verified [19] for a number of programs, including the `403.gcc` benchmark from SPECint 2006.

The output of ISel in Virtual x86 forms the input for our validation experiments. We applied the following combinations of register allocators and optimization levels. Note that these are widely used in production and well-tuned allocators with complex logic.

- `fast -O0`: Default register allocator at `-O0`. This is a local register allocator specifically aiming for speed. Registers are assigned on a greedy basis, and all live registers are spilled at the end of each block.
- `basic -O2`: A register allocator similar to linear scan, but visits live intervals in a different order.
- `greedy -O2`: Default register allocator when optimization is enabled. Similar to the basic allocator but uses more complicated heuristics and supports global live range splitting.
- `pbqp -O2`: A register allocator that reduces the allocation problem to partitioned boolean quadratic programming (PBQP) [13].

For each verification run, we allocated 2 Intel Xeon CPU E7-8837 processors at 2.67GHz and 10GB of memory, with a 120-second timeout on the inference algorithm, and a 60-minute timeout on KEQ.

An overview of the results is shown in Figure 8. Out of a total of 14,542 functions, our Virtual x86 semantics support 12,731 functions. The unsupported features include jump tables, floating point instructions, vector instructions, and indirect jumps. Overall, our tool is able to validate the register allocation for a large fraction of the supported functions, ranging from 88.1% in `basic -O2` to 92.2% in `fast -O0`, with the other two falling in between. Moreover, Figure 9 shows the distributions of LLVM IR LOC, KEQ verification time (mean 58.9s), and inference algorithm time (mean 1.32s) for the tests using `greedy -O2`. These results indicate that our approach is highly successful at validating most of the code in modern, production-quality register allocators *unmodified*, using a fully automatic inference algorithm that requires *no explicit support from the compiler*.

The main reasons of failure for the remaining supported functions are summarized below. We use the statistics from tests on the `greedy` allocator and the percentages are taken with respect to the number of supported functions.

1) Limitation in the inference algorithm: There are in total 895 (7.0%) functions failing due to errors in the inference algorithm, in which case the inference algorithm fail to produce valid invariants at synchronization points. One of the reasons is that the Gen function described in Section III-D employs incomplete heuristics to match instructions, which were not sufficient for some examples. Another 149 tests failed with timeout in the inference algorithm: this is primarily due to our current inefficient implementation, and we expect to solve it by future improvements. The rest of the failed tests in this category can be summarized as edge cases that our implementation is not handling correctly. For example, 72 of them use the x86 register `AH`, while our invariant currently only models sub-registers that are lower bits of the full-sized registers (e.g. `eax`, `ax`, `al`).

Furthermore, some VCs are rejected by KEQ, which accounts for 272 (2.1%) cases. This can be caused by incorrect synchronization points generated by the inference algorithm, or issues in the Virtual x86 semantics.

2) Timeouts and out of memory: 222 (1.7%) tests failed due to timeout or out-of-memory in KEQ. Sometimes this happens during parsing, since $\mathbb{K}$ has a general ambiguity resolving mechanism that does not scale well. In other cases, we may encounter this problem in the SMT solver (for which we currently use Z3 [10]) or symbolic execution due to the complexity of path conditions and invariants.

Overall, the failure rates of the algorithm can be significantly reduced by a more mature implementation. Moreover, an important encouraging sign is that the algorithm is uniformly strong for all four (very different) allocators, achieving roughly 90% coverage in all cases, which bodes well for the usefulness of the algorithm.

B. Evaluation with Real-World LLVM Bugs

We tested KEQ on two bugs found in the `fast` register allocator in LLVM-7.0 and -3.5.

The first bug [2], reported and fixed in LLVM-7.0 (but present in earlier versions), happens when an instruction defines (at least) two registers, with a virtual register definition appearing before a physical register definition in the operand list. In such case, the buggy version of the allocator could potentially use the same physical register for both definitions. An example is shown in Figure 10. The instruction `mulxq` multiplies `rdx` with the third operand, and stores the high and low half of the result to the first and second operands, respectively. When the first two operands are the same, they will both contain the high half of the result. So it is incorrect to assign `rax` to `%vr1`. KEQ detected the problem during symbolic execution of the two versions of code, because the equality constraint for `rax` fails to match between the two versions at the point just before the `retq` instruction. Besides detecting the bug, this information would help guide a compiler developer debugging the problem by pinpointing the program interval in which the faulty allocation occurs.

The second bug [1], reported and fixed in LLVM-3.5, causes incorrect computation of live ranges of physical registers in the input code when the physical registers appear as implicit definitions. An example is shown in Figure 11. Various x86 instructions have implicit operands that do not appear in the assembly syntax but are well-defined in its semantics. In this particular case, the implicit use of `rax` of a signed division instruction `idiv` is ignored, resulting in potentially different division result, and is detected by KEQ in a Z3 query.

Benchmark	Tot	Uns	`fast -O0`			`basic -O2`			`greedy -O2`			`pbqp -O2`		
			P	F	P/S	P	F	P/S	P	F	P/S	P	F	P/S
perlbench	1859	277	1363	219	86.2%	1306	276	82.6%	1315	267	83.1%	1312	270	82.9%
bzip2	100	16	75	9	89.3%	74	10	88.1%	73	11	86.9%	74	10	88.1%
gcc	5572	505	4719	348	93.1%	4634	433	91.5%	4660	407	92.0%	4647	420	91.7%
mcf	24	3	19	2	90.5%	18	3	85.7%	19	2	90.5%	19	2	90.5%
gobmk	2679	192	2421	66	97.3%	2377	110	95.6%	2388	99	96.0%	2385	102	95.9%
hmmer	538	210	288	40	87.8%	286	42	87.2%	287	41	87.5%	286	42	87.2%
sjeng	144	31	100	13	88.5%	99	14	87.6%	99	14	87.6%	100	13	88.5%
libquantum	115	59	53	3	94.6%	53	3	94.6%	53	3	94.6%	52	4	92.9%
h264ref	590	127	397	66	85.7%	382	81	82.5%	386	77	83.4%	386	77	83.4%
omnetpp	2761	351	2191	219	90.9%	1874	536	77.8%	1876	534	77.8%	1877	533	77.9%
astar	160	40	118	2	98.3%	116	4	96.7%	117	3	97.5%	116	4	96.7%
Total	14542	1811	11744	987	92.2%	11219	1512	88.1%	11273	1458	88.5%	11254	1477	88.4%

Fig. 8: Overview of the evaluation on SPECint 2006, where the columns are (Tot)al number of functions, (Uns)upported, (P)assed, (F)ailed, (P)assed/(S)upported. `omnetpp` and `astar` are written in C++ while others are written in C.

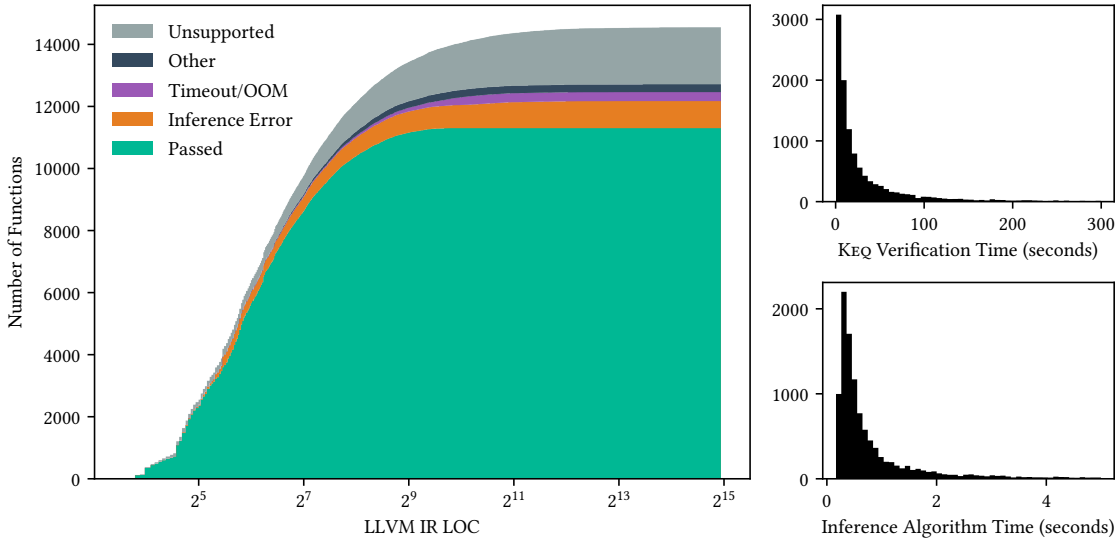

Fig. 9: Statistics for the tests using `greedy -O2`. Left: cumulative histogram of LLVM IR LOC of functions with different results. Right: performance statistics for 11,273 successfully verified functions, where KEQ verification time is cut off at 300 seconds with 439 outliers (max 3295.6s), and inference algorithm time is cut off at 5 seconds with 579 outliers (max 55.4s).

```
.BB0:                          .BB0:
movabsq %vr0,1                 movabsq rax,1
movabsq rdx,1                  movabsq rdx,1
                               movq (frame(1)),rax
mulxq %vr1,rax,%vr0            mulxq rax,rax,rax
retq rax                       retq rax
```

Before register allocation (returns 1) | *After bogus register allocation* (returns 0)

Fig. 10: LLVM Bug 41790 [2]

V. Related Work

Translation Validation was first proposed by Samet [30] and was rediscovered by Pnueli *et al.* [27]. Translation Validation has been often used to assist the compilation process. There are examples for validation of specific optimizations [18], [20], [24], [26], [33], [35]–[38], for discovery of compiler bugs [14], and even for validation of end-to-end compilation [7], [9], [12], [32], [33].

The works that are more closely related to ours target the register allocation and related transformations. Huang *et al.* [15] propose a static analysis approach that guarantees no false alarms and generates informative messages when detecting errors in the output of the allocator. This approach is useful for discovering bugs in the compiler but cannot be

```
.BB0:
COPY %vr1, edi
movl %vr4, 0
movl eax, 0
; sign-extends rax
; to rdx:rax
cqo
movsxq %vr7, %vr1
; implicitly defines
; and uses rax and rdx
idivq (%vr4 + 8 * %vr7)
retq rax
```

Before register allocation

```
.BB0:
movl ecx, 0
movl eax, 0
cqo
movsxq rax, edi
idivq (rcx + 8 * rax)
retq rax
```

After bogus register allocation (`movsxq` overwrites `rax`)

Fig. 11: LLVM Bug 21700 [1]

used in a setting where formal guarantees for the compilation process are needed, since the underlying static analysis does not guarantee correctness when no errors are reported. Moreover, the analysis is not accompanied by a correctness proof. Such proof does not seem trivial as it would require a formal definition of global value numbering, a complex analysis that is part of this system's static analyser.

On the other hand, Rideau *et al.* [28] present another static analysis approach for register allocation that is proven correct in both algorithm and implementation (using Coq), and provides a formal guarantee of correctness for validated compilations. This system is used in the CompCert verified compiler [21] as a replacement for a verified register allocator, since the latter was constrained to a rather naive spilling heuristic that negatively impacted the quality of the compiled code. The Translation Validation approach allows CompCert to use an untrusted and more aggressive spilling algorithm, while maintaining the correctness guarantee for the compilation result. The static analysis used in this work makes similar assumptions to our inference algorithm (i.e. 1-1 correspondence of basic block and non-copying instructions) and manages to validate register allocated code without any symbolic execution. However, the analysis (and its correctness proof) are specific to register allocation, while the KEQ equivalence algorithm (and its correctness proof) can be reused for validation of different transformations[1]. In short, the static analysis in [28] has the advantage that it is a lightweight and fast analysis, but would be impractical to design separately for all the phases of an existing production compiler. Our approach is better suited for existing compilers, by entirely reusing the theoretical results and tools across different transformations.

Nandivada *et al.* [25] proposes RALF, a framework for easy development and evaluation of register allocation algorithms. The framework consists of two languages, MIRA and FORD, and a type system for checking correctness of register allocation. MIRA is an intermediate level language designed to represent programs before register allocation. MIRA programs contain architectural information such as the register file, calling convention, etc., as well as static analysis information such as def-use chains, control flow, etc. FORD is a language for register allocation directives such as directives for spills, register/variable mappings at different program points etc. Finally, the type system is designed to ensure that a type-correct FORD program preserves the values alive in the underlying MIRA program. The proposed framework is mainly intended for fast implementation and testing of register allocation algorithms, since it allows for easy plugging of said implementations into the GCC compilation path and offers built-in validation of the allocator's output. It cannot however be used for translation validation of an existing register allocator within a production compiler, as it only supports allocators that work with the MIRA/FORD intermediate representations.[2] On the other hand, our proposed design focuses on existing compilers and it can be applied to the register allocator of production compilers such as LLVM without modifying the allocator's code.

In a more general setting than register allocation, Necula [26] proposes a Translation Validation system for the GCC compiler that tackles register allocation as one of the many supported transformations. Similar to our approach, the system uses no compiler-generated information but rather employs an intricate inference algorithm to produce equality constraints for each basic block of a function. The inference algorithm uses transfer functions to describe the effect of each basic block, which are generated using symbolic execution of the RTL representation on which the system operates. Various other techniques have also been proposed to check the equivalence of programs in a black-box manner, and potentially across multiple compilation passes. Dahiya *et al.* [9] uses brute-force search of a *joint transfer function graph (JTFG)* and invariants. Churchill *et al.* [7], [33] uses concrete traces to infer alignment and invariants between two x86 programs. Gupta *et al.* [12] uses counterexamples guide the incremental construction of a *product-CFG*. In comparison to the work above, our inference algorithm is more light-weight because it uses a static, intraprocedural dataflow analysis rather than an SMT solver or concrete test cases, since we are able to focus our effort on a specific transformation as opposed to a wide set of optimizations (the proof system in Keq of course uses an SMT solver). By using a general equivalence checker, KEQ, we are essentially refactoring the demanding workload of such automated equivalence proofs (symbolic execution, SMT solvers) out of the inference logic that drives the verification conditions (VC) and needs to take transformation-specific characteristics into account. This is especially important because the VC generator is the only part that must be modified as passes are modified or added to a production compiler, and a simple (yet automatic) VC generator like ours is far easier for

[1]The register allocation specific part of our system, our inference algorithm and the VC generator, is not proven correct but need not to be trusted: KEQ will reject bogus synchronization points as explained in Section II

[2]Of course, a type-checker using the proposed type system could be implemented for the intermediate representation of a target production compiler to be then used for validation of its register allocator.

compiler teams to develop and maintain than one that requires a significant amount of knowledge of formal methods.

VI. Conclusion

In this work we present a Translation Validation system for the register allocation phase of the LLVM compiler, for four different allocators. We based our system on a language- and transformation-independent program equivalence checker, KEQ, and it required no modifications for this work (neither did the Virtual X86 language semantics). The inference of verification conditions is the only part that is specific to register allocation, and this part is fully automated, i.e. it does not require any support from the compiler itself; instead, it operates solely on the input and output programs. In particular, we developed a black-box verification condition generator for use with KEQ that is specific to register allocation, and which uses an inference algorithm for matching live registers between the input and output program. Our prototype system is able to prove the correctness of register allocation for over 88% of supported functions from the SPECint 2006 benchmark on four different production-quality register allocators (some of them quite sophisticated) in LLVM 5.0.2.

References

[1] Register Allocation implicit definition miscompilation. https://bugs.llvm.org/show_bug.cgi?id=21700, 2014.

[2] When same reg is used for output and implicit-def, spill is inserted and overwrites output result. https://bugs.llvm.org/show_bug.cgi?id=41790, 2019.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, page 98–105, New York, NY, USA, 1982. Association for Computing Machinery.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981.

[7] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1027–1040, 2019.

[8] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In Kai Koskimies, editor, *Compiler Construction*, pages 174–187, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[9] Manjeet Dahiya and Sorav Bansal. Black-box equivalence checking across compiler optimizations. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems*, pages 127–147, Cham, 2017. Springer International Publishing.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340, 2008.

[11] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, May 1996.

[12] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. Counterexample-guided correlation algorithm for translation validation. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[13] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with pbqp. In *Joint Modular Languages Conference*, pages 346–361. Springer, 2006.

[14] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 191–201, New York, NY, USA, 2013. ACM.

[15] Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In Kwangkeun Yi, editor, *Static Analysis*, pages 281–300, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, December 2016.

[17] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[18] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: Verified credible compilation for llvm. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 631–645, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. Language-parametric compiler validation with application to llvm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 1004–1019, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 327–337, New York, NY, USA, 2009. ACM.

[21] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[22] LLVM. LLVM Target-Independent Code Generation: Instruction Selection. http://llvm.org/docs/CodeGenerator.html#register-allocator, 2020. Accessed: Friday 12th November, 2021.

[23] LLVM. LLVM Target-independent Code Generator. http://llvm.org/docs/CodeGenerator.html#machine-code-representation, 2020. Accessed: Friday 12th November, 2021.

[24] Kedar S. Namjoshi and Lenore D. Zuck. *Witnessing Program Transformations*, pages 304–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[25] V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 153–169, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[26] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[27] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

[28] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In Rajiv Gupta, editor, *Compiler Construction*, pages 224–243, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[29] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[30] Hanan Samet. *Automatically Proving the Correctness of Translations Involving Optimized Code.* PhD thesis, Stanford, CA, USA, 1975. AAI7525601.

[31] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.

[32] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 471–482, New York, NY, USA, 2013. ACM.

[33] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM*

SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 391–406, New York, NY, USA, 2013. ACM.

[34] SPEC. SPEC CPU 2006 Benchmark. https://www.spec.org/cpu2006/, 2020. Accessed: Friday 12th November, 2021.

[35] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 737–742, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

[37] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM.

[38] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9:2003, 2003.