Spatial and Temporal Decomposition for Faster Translation Validation

BENJAMIN MIKEK, Georgia Institute of Technology, USA
CHATHUR BOMMINENI, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA
THOMAS REPS, University of Wisconsin-Madison, USA

Translation validation is a critical tool in program analysis: when a program P is transformed into a new program P', translation validation asks whether P and P' have the same semantics. It serves as a middle ground between compiler testing and formal verification, capable of proving that a particular run of a compiler produced correct results. However, one bottleneck holds back wider adoption of translation validation: performance. State-of-the-art tools frequently time out or require extensive manual engineering to adapt to specific use cases. In this paper, we propose a new approach to improving the scalability of translation validation by decomposing the problem along two axes. Our primary contribution is a method for harnessing compiler information to extract subprograms whose equivalence result implies equivalence of the overall transformation (the spatial axis). We augment this method by utilizing compiler information to dynamically group transformation passes for validation (the temporal axis). Our evaluation demonstrates that this approach validates 10% of translations that existing approaches fail to validate, and speeds up validation by up to 2.4×.

CCS Concepts: • **Software and its engineering → Formal software verification**.

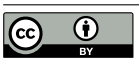
Additional Key Words and Phrases: Translation Validation, Compilers, Program Decomposition

ACM Reference Format:
Benjamin Mikek, Chathur Bommineni, Qirun Zhang, and Thomas Reps. 2026. Spatial and Temporal Decomposition for Faster Translation Validation. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 380 (October 2026), 29 pages. https://doi.org/10.1145/3839512

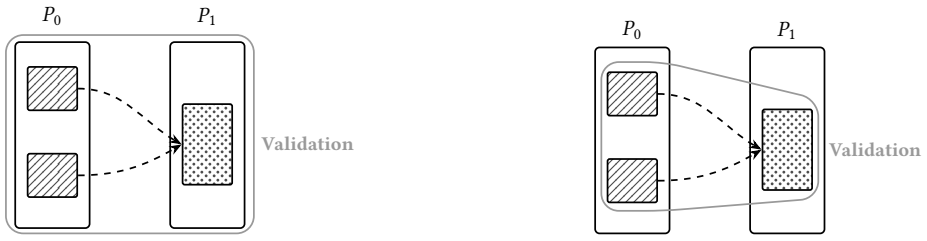
1 Introduction

Translation validation is a key problem in program analysis [32, 33]: given a program P, the goal is to determine whether a transformed program P' preserves the semantics of P. Translation validation proves that the compilation of a particular program was correct; it has been used to detect bugs in production compilers [3, 25] and forms part of the formally verified compiler CompCert [42]. However, wider adoption of translation validation remains far off: it is held back by long running time and frequent timeouts. For example, the state-of-the-art translation validator Alive2 [25], which calls an SMT solver to reason about program semantics, times out on more than half of functions in a realistic benchmark. The root cause of this scalability bottleneck is that existing approaches construct a single monolithic constraint problem that encodes far more complexity than needed. Consider an optimizer that transforms P into P' via a sequence of intermediate programs $P = P_0, P_1, \ldots, P_N = P'$. We can divide the translation validation problem along two axes.

Authors' Contact Information: Benjamin Mikek, Georgia Institute of Technology, Atlanta, GA, USA, bmikek@gatech.edu; Chathur Bommineni, Georgia Institute of Technology, Atlanta, GA, USA, bchathur3@gatech.edu; Qirun Zhang, Georgia Institute of Technology, Atlanta, GA, USA, qrzhang@gatech.edu; Thomas Reps, University of Wisconsin-Madison, Madison, WI, USA, reps@cs.wisc.edu.



ACM 2475-1421/2026/10-ART380
https://doi.org/10.1145/3839512



(a) Existing approach to translation validation along the spatial axis.

(b) Our approach to decomposing translation validation along the spatial axis.

Fig. 1. Existing approaches and our approach along the spatial axis.

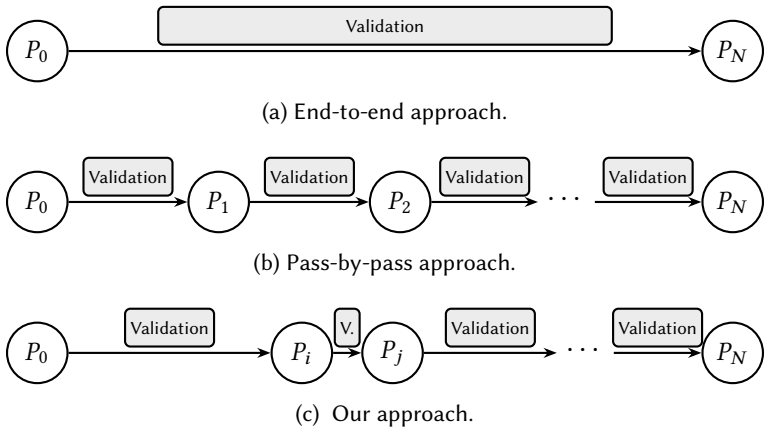
- The **spatial axis** corresponds to the structure of each program: the instructions and control flow graph of any single program P_i (including P and P') in the optimization pipeline.
- The **temporal axis** corresponds to the structure of the optimization pipeline itself: the sequence of programs $P_0, P_1, \ldots, P_N$ produced by successive optimization passes.

Along the spatial axis, a transformation T typically modifies only a fragment of a program P_i—the *redex*—to produce a modified fragment—the *reductum*—in P_{i+1}, as shown in Figure 1a. Instructions outside these sets remain unchanged from P to P', yet existing tools encode the entirety of P and P' into constraints, forcing the solver to reason about code that is irrelevant to the transformation. Checking the equivalence of larger programs increases verification time: in our experiments, we found a strong ranking correlation of $\rho = 0.72$[1] ($p < 1 \times 10^{-10}$) between program size and the running time of the state-of-the-art translation validation tool Alive2.

Along the temporal axis, existing methods take one of two paths. The first approach is *end-to-end* translation validation, shown in Figure 2a. Here, a validator bridges the full gap between P and P' with a single constraint [3, 5, 13, 35]. The second existing approach is the *pass-by-pass* strategy, shown in Figure 2b. Here, a validator checks equivalence after every single transformation [2, 20, 32]. The two approaches lie at opposite ends of the design spectrum. The *end-to-end* approach ignores the pipeline structure of compiler optimizations, calling a solver one time on a pair of structurally dissimilar programs. The *pass-by-pass* strategy, on the other hand, calls a solver many times, but each individual validation spans fewer program differences. In our experiments, the magnitude of changes from P to P' is also highly correlated to validation time ($\rho = 0.51$, $p < 1 \times 10^{-10}$).

In this paper, we propose to improve the scalability of translation validation by decomposing the monolithic translation validation problem along the spatial axis. Our approach is to extract fragments $f \subset P$ and $f' \subset P'$, as shown in Figure 1b, such that the question of equivalence between P and P' is reduced to that of equivalence between f and f'. Because f and f' are subsets of P and P', their semantics are easier to reason about. Our key insight is that f and f' can be constructed by combining information from compiler instrumentation—what instructions were modified—with a lightweight program analysis—what additional instructions are required to make f and f' sensible functions. The approach is sound under the condition that the instrumentation used to track instruction changes is correct; essentially, it trusts the compiler to accurately *report what it did*, but does not assume that the optimizations performed *were correct*. By minimally expanding the trusted foundation in this way, we reduce the burden of checking the semantics of all of P and P'.

[1]We report the Spearman correlation [36], which measures the correlation between rankings, because this quantity strips away the high volatility of absolute SMT solving times while still measuring monotonic relationships. The Spearman correlation has been used in the literature to measure similar phenomena in program verification [6] and SAT solving [39].



(a) End-to-end approach.

(b) Pass-by-pass approach.

(c) Our approach.

Fig. 2. Our approach compared to the two existing modes of Alive2 along the temporal axis.

We augment spatial decomposition with a new set of temporal decomposition heuristics enabled by compiler instrumentation: our approach is to perform validation at a set of intermediate points during compilation, striking a balance between the existing end-to-end and pass-by-pass approaches. Our key insight is that the same information about transformation passes' actions used for spatial decomposition can also be used to tailor the selection of intermediate validation points to each input program. Moreover, spatial and temporal decomposition can be combined on a single run of translation validation. Adding temporal decomposition to spatial decomposition allows the latter to reach its full potential by reducing the number of modified instructions during each transformation while still keeping the number of solver calls small. The combination outperforms either decomposition strategy on its own.

We realize our decomposition strategy as DESTIVA (DEcomposing Spatially and Temporally for translatIon VAlidation), which implements both spatial and temporal decomposition using the existing translation validation tool Alive2 [25] as a foundation. Our evaluation on a large set of diverse benchmark functions shows that DESTIVA outperforms Alive2, validating the translations of 10% of benchmark functions for which Alive2 times out, and achieving a geometric mean speedup of 2.4× for functions where validation is difficult. Portfolio methodology may be needed, however, to combat cases where DESTIVA times out on benchmarks that Alive2 validates. We find that spatial decomposition alone speeds up validation, and we evaluate several temporal decomposition heuristics, finding that combining temporal decomposition with the spatial axis further boosts performance. In sum, we make the following contributions:

- We introduce spatial decomposition for translation validation, and give a method for extracting program fragments whose equivalence implies equivalence of the entire program before and after a transformation.
- We develop several temporal decomposition heuristics that harness compiler instrumentation to strike a balance between the existing *end-to-end* and *pass-by-pass* approaches.
- We implement both spatial and temporal decomposition and find that their combination significantly outperforms the state-of-the-art translation validation tool Alive2.

The rest of this paper is organized as follows. Section 2 gives a motivating example for our approach. Section 3 provides relevant definitions and theoretical background, while Section 4 formally describes our approach. Section 5 describes our experimental results. Section 6 discusses our results and future work, while Section 7 surveys related work and Section 8 concludes.

```
%maxLocal = getelementptr inbounds nuw %
      struct.BtShared, ptr %pBt, i32 0,
      i32 11 ◄
store i16 %conv129, ptr %maxLocal
%28 = load i32, ptr %usableSize126
%sub131 = sub i32 %28, 12 ◄
%mul132 = mul i32 %sub131, 32 ◄
%div133 = udiv i32 %mul132, 255
%sub134 = sub nsw i32 %div133, 23 ◄
%conv135 = trunc i32 %sub134 to i16 ◄
```

(a) An excerpt from the `lockBtree` function P. Instructions affected by the optimization are marked with "◄".

```
define {i32, i32} @frag_pre(ptr  %pBt,
      i32 %28) {
  %maxLocal = getelementptr inbounds nuw
      %struct.BtShared, ptr %pBt, i32 0,
      i32 11
  %sub131 = sub i32 %28, 12
  %mul132 = mul i32 %sub131, 32
  %div133 = udiv i32 %mul132, 255 ; flow
  %sub134 = sub nsw i32 %div133, 23
  %conv135 = trunc i32 %sub134 to i16
  ret { i32, i32 } { i32 %maxLocal, i32
     %conv135 }
}
```

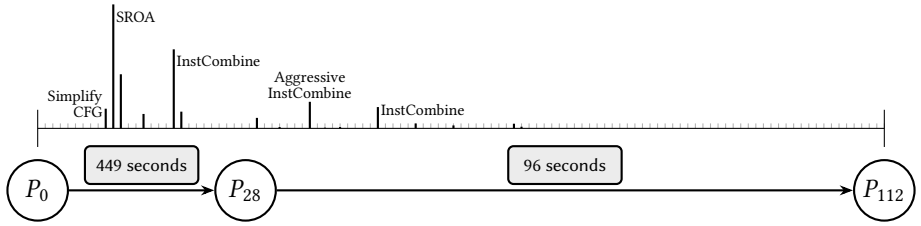
(b) Code of the extracted fragment f before optimization.

```
%maxLocal = getelementptr inbounds nuw
      i8, ptr %pBt, i64 40 ◄
store i16 %conv129, ptr %maxLocal
%32 = load i32, ptr %usableSize126
%sub131 = shl i32 %32, 5 ◄
%mul132 = add i32 %sub131, -384 ◄
%div133 = udiv i32 %mul132, 255
%33 = trunc i32 %div133 to i16 ◄
%conv135 = add i16 %33, -23 ◄
```

(c) P', the excerpt after optimization. Modified and inserted instructions are marked with "◄".

```
define {i32, i32} @frag_post(ptr  %pBt,
      i32 %28) {
  %maxLocal = getelementptr inbounds nuw
      i8, ptr %pBt, i64 40
  %sub131 = shl i32 %28, 5
  %mul132 = add i32 %sub131, -384
  %div133 = udiv i32 %mul132, 255 ; flow
  %33 = trunc i32 %div133 to i16
  %conv135 = add i16 %33, -23
  ret { i32, i32 } { i32 %maxLocal, i32
     %conv135 }
}
```

(d) The post-optimization extracted function f'.



(e) Diagram of optimization passes with LLVM 22.1. Bar height corresponds to the number of instructions changed by the optimization; gray ticks represent passes that made no changes to the function.

Fig. 3. Motivating example for our spatial and temporal decomposition approach with 600-second timeout.

2 Motivating Example

This section gives an example that motivates our spatial decomposition approach and previews our temporal decomposition heuristics. Both existing modes of Alive2 (Figures 2a and 2b) time out for this function, as does spatial decomposition alone. Temporal decomposition reduces validation time to 545 seconds, while the combination of both axes reduces it to just seven seconds.

Code Snippet. We use as an example function `lockBtree` from `SQLite3`. This function consists of 340 LLVM IR instructions across 35 basic blocks; its purpose is to acquire a lock on a B-tree data structure. LLVM's optimizer runs 112 optimization passes on this function, though only 15 make changes to the program, as shown by the black tick marks in Figure 3e.

Existing Methods. Alive2 provides two methods of validating the translation. The first is *end-to-end* translation validation (Figure 2a): this method compares P to P' without considering any intermediate optimizations. End-to-end translation validation requires calling a solver only once, but it does so for two very different programs; Alive2 fails to prove the translation for this function within 600 seconds using end-to-end translation validation.

The second method is *pass-by-pass* translation validation (Figure 2b): this method checks the equivalence of the before-and-after functions after each pass runs. Pass-by-pass validation makes each call to the solver easier, since the two programs are relatively similar, but it still calls the solver 15 times. The 600-second "budget" therefore has to be divided among the calls, and Alive2 also fails to provide a result within 600 seconds using pass-by-pass validation.

The Spatial Axis. Decomposing along the sequence of optimization passes does not yet take full advantage of the structure of the problem: most optimization passes only change small parts of a program. With spatial decomposition, we take advantage of this structure by reducing the translation validation task to only the relevant parts of a program P. Figure 3a shows an excerpt of `lockBtree`'s code; it first does an element-pointer computation, stores a value, and then proceeds to instructions that manipulate the `usableSize126` value loaded from memory. During optimization with LLVM's `instCombine` pass, Figure 3a is transformed into the code given in Figure 3c. In both the original code (Figure 3a) and final code (Figure 3c), "◀" indicates the instructions that are changed by the compiler.

The `getelementptr` instruction is assigned a new constant offset for 64-bit integers, the subtract-multiply pair is transformed into a shift followed by addition, and the subtraction-truncation pair is replaced by a `trunc` instruction followed by addition. To check that this transformation is correct, existing translation validation approaches would reason about the entire contents of P and P', even though several instructions are not affected by the optimization. Figure 3b gives an LLVM IR function `frag_pre` that encapsulates the code in Figure 3a. It takes as input values for the pointer `%pBt` and the register `%28` and returns a pair of values, `%maxLocal` and `%conv135`. Its body consists of all of the instructions modified by the optimization that transformed Figure 3a into Figure 3c. We also add the statement `%div133 = udiv i32 %mul132, 255` to `frag_pre` and `frag_post` because it is a data-flow intermediary between changed instructions: data flows to the `udiv` from a modified instruction (via `mul132`) and flows from it to another modified instruction (via `div133`). The extracted function *does not* include the store or load instructions, even though these instructions would be executed in between modified instructions in the original. It is safe to exclude them and give the result of the load as an argument to `frag_pre` because `store` has no data-flow relationship to the modified instructions, and the load only has data flow *to* a modified instruction, but not *from* one.

Figure 3d gives `frag_post`, a function extracted from Figure 3c in the same way that `frag_pre` was extracted from Figure 3a—Figure 3d *is not* the result of optimizing Figure 3b. We can use an existing translation validation method, such as Alive2, to check whether the program in Figure 3d is equivalent to the program in Figure 3b; in this case, it is. Because the instructions we excluded from `frag_pre` and `frag_post` were not changed by the compiler, the fact that `frag_pre` and `frag_post` have the same semantics is sufficient to conclude that P and P' do too: we have achieved a translation validation result by comparing two simpler functions. In the full code of `lockBtree`, the combination of temporal and spatial decomposition reduces the validation burden dramatically, decreasing the time it takes Alive2 to perform the validation to just seven seconds.

Temporal Decomposition. Along the temporal axis, there is a tradeoff between making many solver calls with easy constraints and making few solver calls with difficult constraints. Alive2's existing modes take either extreme of this tradeoff, but by choosing a temporal decomposition somewhere in between, we can achieve better results. Figure 3e shows a "Cut Point" after the 28th pass; validating the transformation up to the cut point takes 449 seconds (a majority of the program changes fall within this segment), while validating the transformation from the cut point to P' takes 96 seconds. If the translations from P to P_{28} and P_{28} to P' are both correct, then we may conclude that the entire translation from P to P' is correct. Validation still takes approximately 545 seconds, but we can now confirm that the optimizations performed by LLVM were correct.

Challenges. The remaining challenges for realizing temporal and spatial decomposition for translation validation lie in determining which instructions should be included or excluded from the functions that play the role of `frag_pre` and `frag_post` in the example above. This question is non-trivial because the extracted functions must cover all program elements changed by an optimization, but still maintain a minimal size. In Section 4, we give a method for extracting such functions: our strategy is to collect the modified instructions using compiler instrumentation, and then perform a *program chop* [18, 34] for all source-target pairs from the modified set. For temporal decomposition, we develop and test several heuristics based on the results of compiler instrumentation; see Section 4.4.

3 Preliminaries

In this section, we begin by describing how translation validation is carried out with existing tools. We then lay the foundation for our approach with a formal description of program transformations and a summary of how we instrument LLVM to obtain a record of the transformations it performs.

3.1 Translation Validation

The problem of *translation validation* is that of verifying that a given program transformation T preserved the semantics of the input program P; in Section 4, we define in detail what it means to preserve a program's semantics. Take the state-of-the-art translation validation tool Alive2 [25] as an example. For some transformation T, Alive2 works by converting the semantics of P and P' into an SMT constraint, which is then solved by an SMT solver. On the spatial axis, Alive2 generates constraints that encode the entirety of a program P and its optimized version P'—it performs no decomposition along the spatial axis. Along the temporal axis, existing approaches have generally used one of two methods:

- **End-to-end (E–E, Figure 2a):** During end-to-end translation validation, the validator compares the original program P and the final optimized program P' after all passes have finished and attempts to prove that they are equivalent.
- **Pass-by-pass (P–P, Figure 2b):** With a pass-by-pass strategy, calls to the verifier are made after every intermediate transformation, and it concludes that P and P' are equivalent if validation succeeds for each pass individually.

Our Approach. We introduce a decomposition strategy along the spatial axis (the vertical axis of Figure 4). Spatial decomposition works by identifying those instructions from P and P' that have been changed by a compiler transformation T, and reduces the solver's reasoning burden to only those instructions relevant to the transformation performed. Along the temporal axis, the two existing translation validation methods represent extremes of temporal decomposition: the end-to-end method does not decompose the problem at all, treating the compiler as a black box. The pass-by-pass method is temporal decomposition at the lowest granularity: it checks equivalence after every single compiler pass. As we find in Section 5, neither method is universally faster: pass-by-pass validation runs faster for some programs and transformations, while end-to-end validation is faster for others. The two methods represent a tradeoff: pass-by-pass validation requires more solver calls, but generates fewer constraints for each one, while end-to-end validation performs a single solver call with a complex monolithic constraint. The aim of our temporal decomposition technique is to strike a balance between these two extremes by harnessing the same information about instructions modified by T to select which transformations to group together during validation. In Section 4, we describe how we conduct spatial decomposition (Section 4.2) and introduce several new temporal decomposition heuristics (Section 4.4). First, we provide some background relevant to our approach.

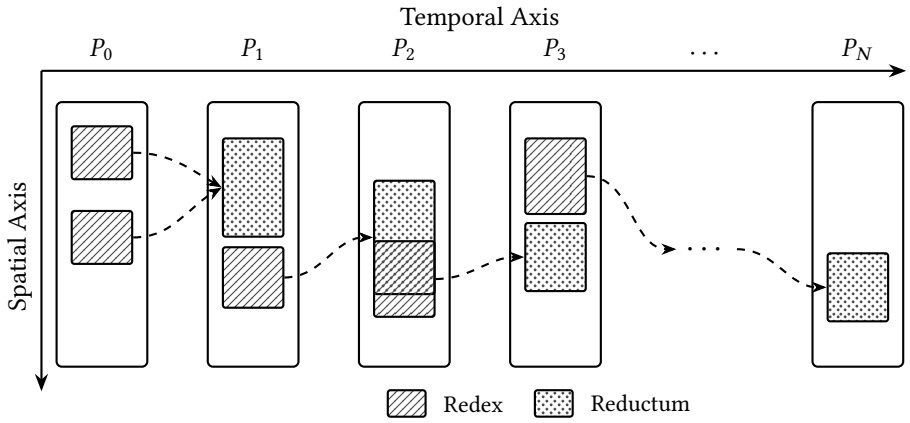


Fig. 4. Overview of program transformations and the spatial and temporal axes of our approach.

3.2 Program Transformations

Figure 4 gives an overview of how a compiler transforms a program along both the spatial and temporal axes. An input program $P = P_0$ with some set of instructions is first optimized by a transformation T_1, producing $P_1 = T_1(P_0)$. A program transformation $T : P \mapsto P'$ is simply a function that converts one program into another. Program transformations consist of one or more of the following atomic building blocks:

- **Removal:** an instruction $i \in P$ is removed so that $i \notin P'$,
- **Insertion:** a new instruction $i \notin P$ is added to P',
- **Modification:** an existing instruction $i \in P$ is modified to produce a new instruction $i' \in P'$.

Any practical program transformation, especially one that preserves semantics, consists of several such operations; some instructions are removed, others are inserted, and some may be modified. In this work, we treat instruction movement (*i.e.*, reordering) as a paired removal and insertion. A transformation T can change the control-flow graph (simplified in Figure 4) by modifying the branch instructions of one or more existing blocks, by inserting a new branch instruction, or by removing a branch instruction and replacing it with a new one. A compiler like LLVM is composed of a sequence of individual transformations (passes) $\mathcal{T} = T_N \circ T_{N-1} \circ \cdots \circ T_1$. The key ingredient for achieving both spatial and temporal decomposition is information about which (and how many) instructions are modified by a particular transformation T; formally, we want the *modified set* for T over an input program P.

Definition 3.1 (Modified Sets). Given a program transformation T as applied to program P to produce transformed program P', we define the

- Removed set $A_{rem}(T, P) \subset P$, the set of instructions that are removed from P during the transformation T, the
- Added set $A_{add}(T, P) \subset P'$, the set of instructions added by T to produce P', and the
- Modified set $A_{mod}(T, P) \subset P \cap P'$, the set of instructions present in both P and P' but modified by T.

We further define $A(T, P) = A_{rem}(T, P) \cup A_{mod}(T, P)$ and $A'(T, P) = A_{add}(T, P) \cup A_{mod}(T, P)$. In the rest of this paper, we use the shorthand A and A' to refer to $A(T, P)$ and $A'(T, P)$, respectively, where T and P are clear.

A and A' contain all of the instructions that are changed by a transformation pass T; A contains those from P that have been removed or modified, and A' contains those from P' that were modified or inserted. We have constructed A and A' so that any instruction not included in either set remains the same across the transformation T, which we state formally in Property 3.1.

Property 3.1. $P \setminus A = P' \setminus A'$.

We assume that LLVM obeys Property 3.1, and make use of it in several proofs in Section 4.2. In order to obtain A and A' in practice, we instrument LLVM.

3.3 Compiler Instrumentation

We obtain A by instrumenting the compiler in a way that allows us to deduce the set of removals, insertions, and modifications performed during an optimization pass T. In particular, before each transformation, we add a tag with a unique integer index to every instruction in P. After the pass has run, we iterate through the function again and assign new tags to each instruction that does not yet have one. After an optimization pass finishes, we are left with two functions P and P', in which each instruction has a tag. Let $tag(i)$ be the tag assigned to an instruction i, and use the notation $i = j$ to mean that instruction i is syntactically equal to instruction j. We construct the three modified sets of Definition 3.1 as follows. From A_{rem}, A_{add}, and A_{mod}, we obtain A and A', which we use extensively in Section 4 to effect both spatial and temporal decomposition:

$$\begin{aligned} A_{rem} &= \left\{ i \in P : \nexists j \in P', \text{tag}(i) = \text{tag}(j) \right\} \\ A_{add} &= \left\{ j \in P' : \nexists i \in P, \text{tag}(i) = \text{tag}(j) \right\} \\ A_{mod} &= \left\{ i \in P : \exists j \in P', \text{tag}(i) = \text{tag}(j) \wedge i \neq j \right\}. \end{aligned} \tag{1}$$

4 Approach

In this section, we first describe our formal model of programs and define a refinement relation between programs along the same lines as that given by Lopes et al. [25]. We then describe our spatial and temporal decomposition techniques and prove that they are correct (Theorems 4.7 and 4.9).

4.1 Definitions

Let $\mathcal{V}$ be a space of values, and define a memory M to be a finite indexable set of such values $M = \{m_1, m_2, \ldots, m_k\} \subset \mathcal{V}^k$; for all programs we describe, we assume k is constant and sufficiently large for all operations performed. We model a program as a function $P : \mathcal{V}^p \times M \times \{\top, \bot\} \to \mathcal{V}^q \times M \times \{\top, \bot\}$; P takes as input p arguments $v_1, v_2, \ldots, v_p$, a memory state M, and an input undefined behavior boolean ub_{in}. It returns a tuple (R, M, ub_{out}) where $R = \{r_1, r_2, \ldots, r_q\}$ is the return value, M is a memory state after execution, and ub_{out} is a boolean indicating whether P encountered undefined behavior during execution. We define programs to take in a boolean undefined behavior flag, which diverges from the treatment in Lopes et al. [25], as a notational convenience so that all programs have a consistent interface for composition. If P is executed on an input where ub_{in} is `false`, it proceeds as normal, returning a `true` ub_{out} result if it encountered undefined behavior during execution. If P is executed on an input with ub_{in} = `true`, then it always produces output where ub_{out} is `true`.

In practice, P models a single function represented in the intermediate representation (IR) of LLVM, and $\mathcal{V}$ consists of LLVM's types (integers in binary, floating-point numbers, *etc.*). We allow P to return multiple values $R = \{r_1, r_2, \ldots, r_q\}$ for ease of notation, though in practice, P returns either a scalar value or a single aggregate struct. The semantics of a program P are given by a control flow graph (CFG) consisting of basic blocks; each basic block contains a straight-line sequence of

instructions $B = \{i_0, i_1, \ldots, i_k\}$ in static single assignment (SSA) form. In this work, we consider only programs that are acyclic, *i.e.*, they may contain branches, but no loops. As we describe in Section 6, it is possible to unroll loops up to a fixed bound to obtain an acyclic program following the approach of Lopes et al. [25], though this cannot guarantee a correct validation result for all loop-containing programs.

An *instruction* i is itself a program; it takes as input the current memory state and a set of SSA variables, and returns a new memory state and a new assignment of SSA variables; it may also trigger undefined behavior. We group instructions into three classes: the first are regular instructions, which take as input SSA variables and define a new SSA variable. Second, there are memory operations: memory loads (reads) read a value from a specified location in memory and store it in an SSA variable, while memory stores (writes) take a value from an SSA variable and write it to a specified location in memory. Finally, there are branch instructions that take SSA variables and direct branching to a new basic block.

Data Flow. We use the notation $i_1 \rightarrow_d i_2$ to mean that there is a data flow from instruction i_1 to i_2. Data flows might occur via SSA variables or via memory. For instructions that operate on SSA variables, $i_1 \rightarrow_d i_2$ if there is a single variable defined by i_1 and used in i_2; SSA form guarantees that there are no intervening defs of the same variable. For memory operations, $i_1 \rightarrow_d i_2$ if i_1 is a write to a memory location ℓ, i_2 is a read from location ℓ, and no memory writes to ℓ occur between i_1 and i_2. We formulate our analysis to allow overapproximating aliasing for memory operations, *i.e.*, we assume that $i_1 \rightarrow_d i_2$ unless we can prove that the address read by i_2 does not alias the address written by i_1. We also assume that $i_1 \rightarrow_d i_2$ if there is any path in the CFG leading from i_1 to i_2 without an intervening write, even though any actual execution would only follow one of those paths. Incomplete alias information might result in including too many instructions in f as defined below, but never too few. In practice, we rely on the information provided by LLVM's existing alias analysis, including "may-alias" relationships.

Equivalence and Refinement. Intuitively, a program transformation is correct if it has preserved the semantics of the original input program P. We adopt the notion of a *refinement relation* used by Lopes et al. [25] as our formal definition of semantics preservation, and restate it here for convenience, first for values, and then for entire programs. Using a refinement relation allows us to account for the fact that the semantics of LLVM's intermediate representation support a few forms of undefined behavior (*poison* values, *undef* values, *noreturn*, etc.). A correct transformation may collapse undefined behavior to a particular concrete behavior, *refining* the program's semantics, but not vice versa. In this work, we do not provide a deep treatment of the semantics of poison and other undefined behavior in LLVM, since it is largely orthogonal to our approach: spatial and temporal decomposition are possible given any formal model of IR semantics. Instead, we state Definition 4.1 and Definition 4.2 in terms of "more defined" and "less defined" values, encompassing poison and all other types of LLVM undefined behavior.

Definition 4.1 (Refinement for Values). A value v is refined by another value v', denoted $v \sqsupseteq v'$, if v' is equally or more defined than v. For scalar values v_1, v_2, $v_1 \sqsupseteq v_2$ if $v_1 = v_2$ or if v_1's value is not defined. For aggregate values, refinement is element-wise.

We next define a notion of refinement for programs, Definition 4.2, based on the refinement of input and output values.

Definition 4.2 (Refinement Relation). A program P' refines a program P, denoted $P \sqsupseteq P'$, if, given inputs I and I' such that $I \sqsupseteq I'$ where $P(I) = (R, M, ub_{out})$ and $P'(I') = (R', M', ub'_{out})$, either ub_{out} is true or $R \sqsupseteq R'$ and $M \sqsupseteq M'$.

4.2 Spatial Decomposition

We now introduce spatial decomposition; here, we take a fixed temporal view and only consider a single transformation—the results we prove hold whether T is a single compiler pass or a composition of many passes. Given a program P and a transformation $T : P \mapsto P'$, our aim is to construct a new program f whose set of instructions is a subset of that of P and captures the changes made to P by T. We achieve this goal by partitioning P into four parts, constructing the new functions, f_{before}, f, f_{ind}, and f_{after} which partition P, and f'_{before}, f', f'_{ind}, and f'_{after}, which partition P'. Figure 5a gives an intuitive overview of the purposes of these four functions. Function f is designed to contain those instructions changed by the transformation T; f_{before} represents operations that produce values used in f, while f_{after} contains uses of values defined in f, and f_{ind} contains instructions with no data flow relationship to f.

We first show that the composition of these functions is equivalent to the original program P, and a symmetric result for P' (Lemmas 4.3 and 4.4). We next prove that all of the functions other than f (f_{before}, f_{after}, f_{ind}) remain the same between P and P' (Lemma 4.6). Combined, these results mean that if $f \sqsupseteq f'$, then $P \sqsupseteq P'$ (Theorem 4.7), achieving the overall goal of smaller programs whose equivalence implies the equivalence of P and P'. We present the construction of f_{before}, f, f_{ind}, and f_{after}, and the proofs of Lemmas 4.3, 4.4 and 4.6 and Theorem 4.7 taking A and A' to be the ground truth. Because A and A' originate from instrumentation of the very compiler whose translations we are attempting to validate, each theorem is conditioned on the correctness of A and A'. In Section 4.3, we describe this potential threat to the soundness of spatial decomposition and the ways in which we combat it.

Defining f and f'. The first step is to construct f; intuitively, f represents the part of the program changed by T. Function f takes as arguments any values used by its instructions, and outputs an aggregate of all of the values defined by its instructions but used elsewhere. Formally, we begin with the set of modified instructions A. The instructions of A are not quite enough to make up the body of f, though. For example, if an instruction i outside f needed to both define and use a value modified inside f, should i be executed before or after f? We solve this problem by expanding f to ensure that no such instructions exist; there are two classes of "problematic" instructions.

First, we add instructions whose data-flow relationships to instructions in A are changed by the transformation T; we call this set B. Adding this class of instructions is necessary to properly partition P and P' (see Lemma 4.6). B consists of the following (we consider only instructions $i \notin A$):

- $i \in B$ if it *gained a data flow to* A': there is no instruction $j \in A$ such that $i \rightarrow_d j$ but there is an instruction $j' \in A'$ such that $i \rightarrow_d j'$;
- $i \in B$ if it *lost a data flow to* A: there is an instruction $j \in A$ such that $i \rightarrow_d j$ but there is no instruction $j' \in A'$ such that $i \rightarrow_d j'$;
- $i \in B$ if it *gained a data flow from* A': there is no instruction $j \in A$ such that $j \rightarrow_d i$ but there is an instruction $j' \in A'$ such that $j' \rightarrow_d i$;
- $i \in B$ if it *lost a data flow from* A: there is an instruction $j \in A$ such that $j \rightarrow_d i$ but there is no instruction $j' \in A'$ such that $j' \rightarrow_d i$.

The dataflow relationship of an instruction i to another instruction j can change either because the instruction i is changed by T, j is changed by T, or a "third party" instruction l with data flow from i and to j changes. By the construction of A and A', any change to data flow relationships must flow through A or A', meaning that B encompasses all instructions that gain or lose *any* data flow during the transformation T. We state this formally in Property 4.1.

Property 4.1. *If $i \notin \{A \cup A' \cup B\}$, then i neither gains nor loses data flow to or from any instruction j in $P \cup P'$.*

The next class of instructions is those with two-way data flows:

$$C = \{i \in P \setminus A : \exists i_1, i_2 \in A \cup B, i \rightarrow_d i_1 \wedge i_2 \rightarrow_d i\} \\ \cup \{i \in P' \setminus A' : \exists i_1, i_2 \in A' \cup B, i \rightarrow_d i_1 \wedge i_2 \rightarrow_d i\}. \tag{2}$$

We express the two conjuncts that make up C based on $P \setminus A$ and $P' \setminus A'$ for clarity, although these two sets are in fact equal by Property 3.1. The set C is a *program chop* [18, 34] with respect to all pairs of instructions in $A \cup B$: it contains all instructions that have a forward data flow dependence to an instruction in $A \cup B$ and a backward data flow dependence from an instruction in $A \cup B$. C is constructed to be closed under two-way data flow; we state this formally in Property 4.2.

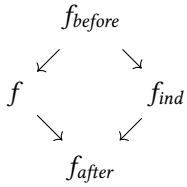
Property 4.2. *If $i \notin \{A \cup A' \cup B \cup C\}$ then there is no j, l in A or A' such that $j \rightarrow_d i \rightarrow_d l$.*

We call the union of these sets $S = A \cup B \cup C$; S will make up most of the body of the function f. With S in hand, we now construct the three additional sets of instructions S_{before}, S_{after}, and S_{ind}. Roughly, these sets correspond to instructions that, respectively, produce data flowing into S, consume data flowing out of S, and are independent of S. Formally, we define the set of instructions S_{before} so that $i \in S_{before}$ if $i \notin S$ and there exists an instruction $j \in S$ such that $i \rightarrow_d j$. Similarly, define S_{after} to be the set of instructions i such that $i \notin S$ and there exists $j \in S$ such that $j \rightarrow_d i$. Finally, let S_{ind} contain $i \notin S$ if, for all $j \in S$, neither $i \rightarrow_d j$ nor $j \rightarrow_d i$ holds.

Finally, we add one final class of instructions: branches.Define $Branch(Y)$ for a given set of instructions Y as the set of all branch instructions $i \in P$ for which $\exists j \in Y$ such that $i \rightarrow_d j$. Intuitively, instruction i will determine whether or not instruction j is executed in P, so we need to include it when constructing a function out of the instructions in Y. We then augment each set to include branches: define $Q = S \cup Branch(S)$ and the analogous sets Q_{before}, Q_{after} and Q_{ind}.

Next, we must define the inputs and outputs of f. Let V_Q be the set of SSA variables and function parameters (of P) used by an instruction in Q but not defined in Q, and W_Q be the set of SSA variables used by an instruction $i \in P$ and defined in Q—we define $V_{Q'}$ and $W_{Q'}$ symmetrically for A'. We can now define $f : \mathcal{V}^{|V_Q \cup V_{Q'}|} \times M \times \{\top, \bot\} \rightarrow \mathcal{V}^{|W_Q \cup W_{Q'}|} \times M \times \{\top, \bot\}$. The semantics of f are defined by the instructions in Q: we interpret the input as an assignment to the variables of V_Q and execute the instructions of Q in the same order, through a path in the CFG, as they would be executed in P. A branch instruction from $Branch(S)$ that was added to Q but is not in S, may have a destination label pointing to a basic block not included in Q. In this case, we replace that label with its nearest successor in Q. We define $f' : \mathcal{V}^{|V_Q \cup V_{Q'}|} \times M \times \{\top, \bot\} \rightarrow \mathcal{V}^{|W_Q \cup W_{Q'}|} \times M \times \{\top, \bot\}$ analogously: its body is $Q' = S' \cup Branch(S')$, with instructions ordered as in P'. If a value appears in the signature of f but was only used or defined in f', or vice versa, f simply ignores it.

Defining f_{before}, f_{after}, and f_{ind}. Now we will define the functions f_{before}, f_{after}, and f_{ind}. Function f_{ind} gathers together all instructions that are independent of S with respect to data flow. Let V_{ind} be the set of SSA variables and function parameters used by an instruction in Q_{ind} but not defined in Q_{ind}, and W_{ind} be the set of SSA variables used by an instruction $i \in P$ (or, equivalently, $i \in P'$) and defined in Q_{ind}. Then we define $f_{ind} : \mathcal{V}^{|V_{\text{ind}}|} \times M \times \{\top, \bot\} \rightarrow \mathcal{V}^{|W_{\text{ind}}|} \times M \times \{\top, \bot\}$ to have body Q_{ind}. We define f_{before} and f_{after} in an analogous way. Let $f_{before} : \mathcal{V}^p \times M \times \{\top, \bot\} \rightarrow \mathcal{V}^{\left|V_Q \cup V'_Q \cup V_{\text{ind}}\right|} \times M \times \{\top, \bot\}$ have the semantics defined by instructions in Q_{before} executed in the same order as in P. Also, let $f_{after} : \mathcal{V}^{\left|W_Q \cup W'_Q \cup W_{\text{ind}}\right|} \times M \times \{\top, \bot\} \rightarrow \mathcal{V}^q \times M \times \{\top, \bot\}$ be the function with the semantics defined by Q_{after}. We also define the analogous functions for P': f'_{before} contains instructions with data flows to S', f'_{after} contains instructions with data flows from S', and f'_{ind} contains all remaining instructions from P', along with the required branch instructions for each set. Figure 5a summarizes the relationships among the four functions.



(a) Lattice representing the required partial ordering of composition.

f	Instructions modified by the compiler and the data flow chop surrounding them
f_{before}	Instructions with data flow to instructions in f
f_{after}	Instructions with data flow from instructions in f
f_{ind}	Instructions with no data flow to or from f

(b) Description of decomposed functions.

Fig. 5. Summary of our breakdown of P into f, f_{before}, f_{after} and f_{ind}.

Now that we have defined the four functions, we must show that they split up P correctly, *i.e.*, we want the composition $f_{after} \circ f_{ind} \circ f \circ f_{before}$ to be equivalent to P. In order to show this, we first prove a useful intermediate fact, Lemma 4.3, which follows from the construction of the sets S, S_{before}, S_{after}, and S_{ind}.

Lemma 4.3 (Partition of P). *$\{S, S_{before}, S_{after}, S_{ind}\}$ is a partition[2] of the instructions in P. Similarly, $\{S', S'_{before}, S'_{after}, S'_{ind}\}$ is a partition of the instructions in P'.*

Proof. Let $i \in P$ be arbitrary. If $i \notin S$, then there are four options. It may be that i has a data flow to one or more instructions in S, in which case $i \in S_{before}$, or i may have data flow from one or more instructions in S, in which case $i \in S_{after}$. i may have data flow neither from S nor to S, in which case $i \in S_{ind}$, and i is not a member of S_{after} or S_{before}, by definition. Finally, since, $i \notin S$, from Property 4.2, i cannot have data flows both from and to S. A symmetric argument holds for the P' case. □

We now turn to the main result about the four functions we have defined and show that the composition $f_{after} \circ f_{ind} \circ f \circ f_{before}$ is equivalent to P; we could also choose the order $f_{after} \circ f \circ f_{ind} \circ f_{before}$ without loss of generality. The proof of this fact is surprisingly complex; the central idea is that we suppose that both P and the composition are run with some input I, showing that the same instructions are executed in the same order, or that the order of their execution does not matter, on both sides. Showing the latter requires the case-by-case analysis summarized in Table 1.

Lemma 4.4 (Correctness of Decomposition). *$P = f_{after} \circ f_{ind} \circ f \circ f_{before}$ and $P' = f'_{after} \circ f'_{ind} \circ f' \circ f'_{before}$.*

Proof. Let some input I be arbitrary. By Lemma 4.3, any instruction $i \in P$ appears in the body of exactly one of f, f_{before}, f_{after}, or f_{ind}, so the set of executed instructions is the same for $P(I)$ and for $f_{after}(f_{ind}(f(f_{before}(I))))$. Now we only need to show that the instructions are executed in the correct order. Let $i_1, i_2 \in P$ such that, on input I, i_1 is executed before i_2. First, let us assume that both i_1, i_2 belong to one of the sets S, S_{before}, S_{after} or S_{ind}. There are 16 possible cases, summarized in Table 1. If i_1 and i_2 are in the same set (the four "Trivial" cases in Table 1), then, since we suppose that each set preserves the order from P, i_1 and i_2 are executed in the correct order. In addition, if i_1 and i_2 are in different sets where the order of the composition causes i_1 to come before i_2 (*e.g.* $i_1 \in S_{before}$ and $i_2 \in S$), then they are also executed in the same order in P and in the composition (six "Composition" cases in Table 1). We now consider the remaining six cases. Since i_1 is executed

[2]We diverge slightly from the classical definition of a partition and allow S_{before}, S_{after}, and S_{ind} to be the empty set. Formally, some subgroup of the four that includes S itself forms a partition; S should be non-empty for any programs of interest.

Table 1. Possible cases for i_1, i_2 in the proof of Lemma 4.4. For the "Trivial" cases, the ordering of instructions is preserved trivially. For the "Composition" cases, the order of instructions is preserved by the order of composition. In the "No Dataflow" cases, the order of instructions does not matter, because we prove that there cannot be any dataflow between i_1 and i_2.

$i_1 \downarrow, i_2 \rightarrow$	S_{before}	S	S_{ind}	S_{after}
S_{before}	Trivial	Composition	Composition	Composition
S	No Dataflow	Trivial	Composition	Composition
S_{ind}	No Dataflow	No Dataflow	Trivial	Composition
S_{after}	No Dataflow	No Dataflow	No Dataflow	Trivial

before i_2 in P, there is no data flow $i_2 \rightarrow_d i_1$. If there is also no data flow $i_1 \rightarrow_d i_2$, then executing i_1 and i_2 in either order produces the same result. Thus, we need only consider the case where there is a data flow $i_1 \rightarrow_d i_2$; there are six such cases ("No Dataflow" cases in Table 1), and all of them lead to a contradiction of Lemma 4.3, meaning such a dataflow cannot exist and the reordering of the instructions is valid.

- $i_1 \in S_{ind}, i_2 \in S$: Since $i_2 \in S$ and $i_1 \rightarrow_d i_2$, by the definition of S_{before}, $i_1 \in S_{before}$. This is a contradiction;
- $i_1 \in S_{after}, i_2 \in S$: Since $i_2 \in S$ and $i_1 \rightarrow_d i_2$, by the definition of S_{before}, $i_1 \in S_{before}$. This is a contradiction;
- $i_1 \in S, i_2 \in S_{before}$: Since $i_1 \in S$ and $i_1 \rightarrow_d i_2$, by the definition of S_{after}, $i_2 \in S_{after}$. This is a contradiction;
- $i_1 \in S_{ind}, i_2 \in S_{before}$: Since $i_2 \in S_{before}$, by definition, $\exists j \in S$ such that $i_2 \rightarrow_d j$. Since $i_1 \rightarrow_d i_2$, we get $i_1 \rightarrow_d j$, which means by the definition of S_{before}, $i_1 \in S_{before}$. This is a contradiction;
- $i_1 \in S_{after}, i_2 \in S_{before}$: Since $i_2 \in S_{before}$, by definition, $\exists j \in S$ such that $i_2 \rightarrow_d j$. Since $i_1 \rightarrow_d i_2$, we get $i_1 \rightarrow_d j$, which means by the definition of S_{before}, $i_1 \in S_{before}$. This is a contradiction;
- $i_1 \in S_{after}, i_2 \in S_{ind}$: Since $i_1 \in S_{after}$, by definition, $\exists j \in S$ such that $j \rightarrow_d i_1$. Since $i_1 \rightarrow_d i_2$, we get $j \rightarrow_d i_2$, which means by the definition of S_{after}, $i_2 \in S_{after}$. This is a contradiction.

We must also consider the case where $i_1 \rightarrow_d i_2$ and one or both of i_1, i_2 belong to $Branch(S)$, $Branch(S_{before})$, $Branch(S_{after})$ or $Branch(S_{ind})$. As before, if i_1, i_2 belong to the same function or if the order of composition causes i_1 to be executed before i_2, they are executed in the same order as they would be in P. In the case that i_1 and i_2 belong to different functions, i_1 cannot be a branch instruction since *Branch* would include a copy of i_1 in the same function as i_2. The remaining cases are similar to the six "No Dataflow" cases from above. For example, one of the cases is $i_1 \in S_{ind}$ and $i_2 \in Branch(S)$, meaning $\exists j \in S$ such that $i_2 \rightarrow_d j$. Then $i_1 \rightarrow_d j$, which contradicts $i_1 \in S_{ind}$. Similar arguments apply to the rest of the cases. □

Most of the "heavy lifting" needed to show the correctness of spatial decomposition is in the proof of Lemma 4.4, but two more intermediate results are required for the main theorem. Lemma 4.5 is an intuitive fact about refinement relations: if refinements of two functions are composed, then the result is a refinement of the composition. Lemma 4.6 states that the functions other than f do not change between P and P'; this is a straightforward consequence of Property 3.1.

Lemma 4.5 (Compositionality of Refinement Relations). *Suppose we have four functions, ϕ, ϕ', ψ, ψ' such that $\phi \sqsupseteq \phi'$ and $\psi \sqsupseteq \psi'$. Then $\phi \circ \psi \sqsupseteq \phi' \circ \psi'$.*

Proof. Let I, I' be arbitrary inputs to ψ, ψ' and $I \sqsupseteq I'$. By Definition 4.2, $\psi(I) \sqsupseteq \psi'(I')$. Using this and $\phi \sqsupseteq \phi'$, we get $\phi(\psi(I)) \sqsupseteq \phi'(\psi'(I'))$, hence $\phi \circ \psi \sqsupseteq \phi' \circ \psi'$. □

Lemma 4.6 (Equality Modulo f). *$f_{ind} \sqsupseteq f'_{ind}$, $f_{before} \sqsupseteq f'_{before}$, and $f_{after} \sqsupseteq f'_{after}$. In fact, $f_{ind} = f'_{ind}$, $f_{before} = f'_{before}$, and $f_{after} = f'_{after}$*

Proof. By Property 3.1 and the definition of S, $P \setminus S = P' \setminus S'$. Then, by Lemma 4.3, we have that $S_{before} \cup S_{after} \cup S_{ind} = S'_{before} \cup S'_{after} \cup S'_{ind}$. First, we claim that $S_{before} = S'_{before}$ and provide a proof by contradiction. Suppose we have an instruction i such that $i \in S_{before}$ and $i \notin S'_{before}$. This means that $\exists j_1 \in S$ such that $i \rightarrow_d j_1$ and $\nexists j_2 \in S'$ such that $i \rightarrow_d j_2$, i.e. $i \in P \setminus \{A \cup B\}$ and i lost data flow to $j_1 \in P$. This is a contradiction of Property 4.1, so $i \in S_{before}$ implies that $i \in S'_{before}$.

By an analogous argument, if $i \in S'_{before}$, then $i \in S_{before}$, meaning $S_{before} = S'_{before}$. A similar argument also applies to instructions in A and A', meaning that $S_{after} = S'_{after}$. Finally, since $S_{before} \cup S_{after} \cup S_{ind} = S'_{before} \cup S'_{after} \cup S'_{ind}$, $S_{before} = S'_{before}$, $S_{after} = S'_{after}$, and there is no overlap between S_{before}, S_{after}, and S_{ind} by Lemma 4.3, it must be that $S_{ind} = S'_{ind}$. Next, since an instruction in S_{before}, S_{after} or S_{ind} does not belong to S, by Property 4.1, it does not gain or lose any dataflow going from P to P'. This means that $Branch(S_{before}) = Branch(S'_{before})$, $Branch(S_{ind}) = Branch(S'_{ind})$ and $Branch(S_{after}) = Branch(S'_{after})$. And consequently $Q_{before} = Q'_{before}$, $Q_{ind} = Q'_{ind}$ and $Q_{after} = Q'_{after}$. Thus, f_{ind} and f'_{ind}, f_{before} and f'_{before}, and f_{after} and f'_{after} each have the same set of body instructions, and each executes the instructions in the same order as in P or P' by construction. Therefore, $f_{ind} = f'_{ind}$, $f_{before} = f'_{before}$, and $f_{after} = f'_{after}$; if each function pair is equal, then they fulfill Definition 4.1 and a refinement relation exists for each too. □

We are now ready to put the pieces together and prove the final correctness result, Theorem 4.7; this theorem guarantees that a translation validation result for f and f' is sufficient to draw a conclusion about P and P'.

Theorem 4.7. *If $f \sqsupseteq f'$, then $P \sqsupseteq P'$.*

Proof. By Lemma 4.4, $P = f_{after} \circ f_{ind} \circ f \circ f_{before}$ and $P' = f'_{after} \circ f'_{ind} \circ f' \circ f'_{before}$ (this is independent of whether $f \sqsupseteq f'$). By Lemma 4.6, we have $f_{ind} \sqsupseteq f'_{ind}$, $f_{before} \sqsupseteq f'_{before}$, and $f_{after} \sqsupseteq f'_{after}$. We assumed that $f \sqsupseteq f'$. By applying Lemma 4.5 three times, we have $P \sqsupseteq P'$. □

4.3 Soundness and Completeness

In this section, we discuss the implications of Theorem 4.7 and the use of compiler instrumentation on the soundness and completeness of our decomposition strategies. Spatial decomposition is incomplete because the correctness of an optimization may depend on facts about its surroundings, but it is sound on the condition that A and A' accurately reflect all changes made by the compiler.

Completeness. Theorem 4.7 gives us a one-sided guarantee of correctness: if the extracted function f is refined by the transformation T, then we may conclude that P is also refined by the transformation. Unfortunately, the inverse of Theorem 4.7 does not hold: it may be that $f \not\sqsupseteq f'$ but $P \sqsupseteq P'$. This can occur because the transformation T was correct only based on invariants over parts of the program not included in A. For example, suppose that f and f' take as an argument the original program variable v, and that, in the original programs P and P', the value v is loaded from a known-valid memory location. Then Alive2 might conclude that the input $v =$ `poison` causes f' not to refine f, even though a global program view shows that v cannot be `poison`. We address this possible source of false positives by underapproximating. In particular, if Alive2 is able to prove that $f \sqsupseteq f'$, then we conclude that $P \sqsupseteq P'$. Otherwise, we expand f to encompass all of P and run Alive2's verifier again—this may lose time, but does not harm correctness. In other words, it means that spatial decomposition is not complete.

In our experiments on a large dataset of realistic functions, DESTIVA only reported false alarms on 4% (120/3,025), each of which was handled by reverting to whole-program translation validation. We also take a few practical steps to reduce the frequency of false positives. These heuristics are based on a useful invariant of our decomposition: expanding A to add more instructions maintains correctness. Formally, the result of Lemma 4.4 requires that A contain all modified instructions, but it still holds if arbitrary unmodified instructions are added to A. These additions, though, must be done to A rather than S, and the sets A, B, and C must be updated based on the added instructions. In practice, we augment A with additional `alloca` instructions, and mark any pointer arguments of f with the `nonnull` attribute if they originate from allocations made within P. A few values defined in f_{before} can also be guaranteed to be non-poison; we mark these arguments with attributes to reduce Alive2's false alarms as well. Cases where restrictions on arguments of f originating outside f lead to false alarms are analogous to Alive2's lack of support for interprocedural optimizations.

Soundness. Theorem 4.7 proves the soundness of spatial decomposition, but it is subject to one assumption—a violation of which would represent a threat to soundness—namely, that the results of compiler instrumentation, A and A', accurately reflect the actions performed by the compiler. Because translation validation aims to validate the actions of the compiler, this is potentially circular reasoning. To reduce the threat to soundness posed by incorrect compiler instrumentation, we follow the basic principle of trusting LLVM to accurately record *what* it did, but not that what it did was *correct*. The instrumentation described in Section 3.3 inserts tags on every instruction after every transformation pass, and is implemented in LLVM's generic after-pass and before-pass hooks; it is designed to be independent of any (potentially buggy) optimization pass. The only class of LLVM bugs that could disturb the correctness of A and A' would be those that strip or augment custom-tagged metadata attached to compiler instructions; such a bug could not arise from a normal change affecting the semantics of program optimization.

4.4 Temporal Decomposition

We next turn our attention to decomposition along the temporal axis; here, the aim is to select a set of intermediate programs, *cut points* in the sequence of transformations, whose refinement relations imply that $P \sqsupseteq P'$. The correctness argument is far simpler than in the case of spatial decomposition. We first prove Lemma 4.8, a simple lemma that follows from Definition 4.2.

Lemma 4.8. *Let ϕ, ϕ', and ϕ'' be programs, if $\phi \sqsupseteq \phi'$ and $\phi' \sqsupseteq \phi''$, then $\phi \sqsupseteq \phi''$.*

Proof. Let v, v', and v'' be values such that $v \sqsupseteq v'$ and $v' \sqsupseteq v''$. Then we claim that $v \sqsupseteq v''$. We will consider the possible cases. First, if v is undefined or a poison value, then by Definition 4.1, $v \sqsupseteq v''$. Second, if $v \in \mathcal{V}$, then $v' \in \mathcal{V}, v = v'$ and consequently $v'' \in \mathcal{V}, v' = v''$. So $v = v''$ and hence $v \sqsupseteq v''$. Now, let I and I' be inputs to ϕ, ϕ', and ϕ'' such that $I \sqsupseteq I'$. Since $I \sqsupseteq I$, by Definition 4.2, $\phi(I) \sqsupseteq \phi'(I)$. Similarly, since $I \sqsupseteq I'$, $\phi'(I) \sqsupseteq \phi''(I')$. Using the previous result, we get $\phi(I) \sqsupseteq \phi''(I')$. Hence, by Definition 4.2, $\phi \sqsupseteq \phi''$. □

Next, define a sequence of *cut points* $C = \{c_1, c_2, \ldots, c_k\}$ such that $0 \leq c_i \leq N$ for all i. A pair of cut points c_i, c_j defines a *validation segment*, that is, we can verify refinement for the pair of intermediate programs at the cut points: $P_{c_i} \sqsupseteq P_{c_j}$. We require that the set of cut points always include the start and end of the optimization sequence, *i.e.*, $c_1 = 0$ and $c_k = N$. We now prove that validation for every segment derived from the cut points is sufficient to show $P_0 \sqsupseteq P_N$.

Theorem 4.9. *Given cut-points $C = \{c_1, c_2, \ldots, c_k\}$ such that $c_1 = 0$ and $c_k = N$, if for all $i < k$, $P_{c_i} \sqsupseteq P_{c_{i+1}}$, then $P_0 \sqsupseteq P_N$.*

Proof. By induction over i with Lemma 4.8. □

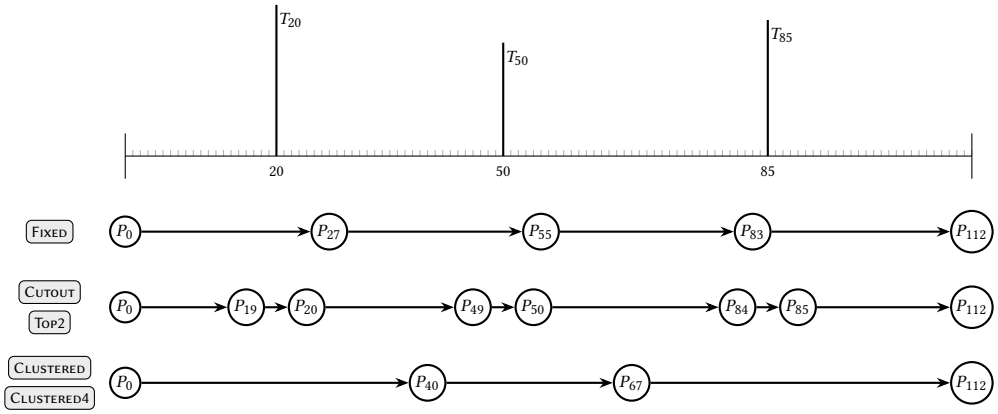


Fig. 6. Summary of temporal decomposition heuristics.

The set of cut points C defines a temporal decomposition; Theorem 4.9 guarantees that performing translation validation with these points preserves correctness. As in the case of spatial decomposition, the inverse of Theorem 4.9 does not hold, at least in theory. A counterexample would consist of two segments between cut points where the first segment's transformation is not a refinement, and the second segment fixes the error. Such examples are rare in practice since LLVM's optimization passes are constructed to each be correct independently; moreover, this source of incompleteness is shared with the existing pass-by-pass method.

Heuristics. We develop five heuristics for temporal decomposition that harness the information we gain from compiler instrumentation to take a middle road between the pass-by-pass and end-to-end approaches of existing work. Figure 6 gives a visual overview of the approaches. We design all five heuristics to result in a similar number of validation segments; for instance, the limits on Top2 and Clustered4 result in five and four verification segments, respectively. This design reduces any effect of segment number as an independent variable in our experiments; as shown in Table 6, all of the heuristics result in an average number of validation segments between 3.5 and 5.0. For all but the Fixed heuristic, we make use of the average number of modified instructions over all passes that change the function, *i.e.*, $z = avg\,(\{|A| \text{ for } T_i : i \leq N \text{ and } |A| > 0\})$.

- Fixed: A set of four segments that evenly divide the optimization sequence. $C = \left\{\frac{N}{4}, \frac{N}{2}, \frac{3N}{4}\right\}$.
- Cutout: Select those passes that make an above-average magnitude change. Then, place cutpoints around each such pass so that it is validated standalone. If $T_{i_1}, T_{i_2}, \ldots, T_{i_n}$ are such that $|A| > z$, then $C = \{i_1 - 1, i_1, i_2 - 1, i_2, \ldots, i_n - 1, i_n\}$.
- Top2: The same as Cutout, but the number of selected passes is limited to the two largest changes. This results in five or fewer (and always an odd number of) segments, as in Figure 6.
- Clustered: As in Cutout, select those passes $T_{i_1}, T_{i_2}, \ldots, T_{i_n}$ that make an above-average change to P. Then create maximal segments containing each such pass; cut points lie at the midpoints between important passes: $C = \left\{\frac{i_1+i_2}{2}, \frac{i_2+i_3}{2}, \ldots, \frac{i_{n-1}+i_n}{2}\right\}$.
- Clustered4: The same as Clustered, but the number of selected passes—and therefore segments—is limited to four.

Each heuristic harnesses the information gained from compiler instrumentation to perform temporal decomposition that creates a middle ground between the pass-by-pass and end-to-end approaches; in Section 5, we evaluate how each performs.

Interaction with Spatial Decomposition. The temporal decomposition heuristics we introduce complement the spatial decomposition strategy described in Section 4.2. In particular, reducing the number of instructions changed during a composed transformation (*i.e.* decreasing $|A|$ and $|A'|$ by grouping together fewer transformation passes) makes f and f' smaller, thereby reducing the burden of solver reasoning. Pass-by-pass translation validation minimizes the size of f and f', but incurs the cost of solver reasoning and the overhead of spatial decomposition more often. As we find in Section 5.3, combining our tailored temporal decomposition heuristics with spatial decomposition outperforms spatial decomposition alone.

5 Evaluation

We evaluated DESTIVA on a set of standard benchmarks, and compared its performance to two existing modes of Alive2. Our key findings are:

- Spatial decomposition alone is able to speed up translation validation for benchmark functions where original validation time was high by as much as 1.3×; for easier functions, the overhead of analysis becomes significant.
- Temporal decomposition alone can speed up translation validation by up to 1.9× compared to existing methods. The most effective heuristics are Clustered and Clustered4.
- Combined, spatial and temporal decomposition achieves speedups up to 2.4× compared to the existing pass-by-pass translation validation method, exceeding either decomposition on its own. While the combined approach validates 298 benchmarks on which existing approaches time out, portfolio methodology is required to offset a number of functions where validation slows down with DESTIVA.

Implementation. DESTIVA's implementation consists of two parts. First is the instrumentation of LLVM source code to obtain the sets A_{rem}, A_{add}, and A_{mod}. Our instrumentation follows the strategy described in Section 3.3, and modifies LLVM's pre- and post-pass callback hooks (`runBeforePass` and `runAfterPass`) to insert index tags as custom metadata on each instruction. We also instrumented the low-level LLVM API operations that change the location of an instruction within a function (`moveBefore`, `moveAfter`, *etc.*) so that they replace an instruction's existing tag with a new unique one.

Second is the core of DESTIVA's spatial and temporal decomposition strategies, which are implemented in C++ as an extension of the existing translation validation tool Alive2 [25]. DESTIVA allows users to enable spatial decomposition and choose a temporal decomposition heuristic via terminal flags. The implementation also depends on existing LLVM interfaces (*e.g.*, `MemorySSA`) to compute data-flow relationships between instructions. We do not rely on any optimization passes to reduce the threat of bugs leaking into instrumentation results. As we discuss in Section 6, we generally assume that our instrumentation accurately reflects what LLVM does, but not that any of its transformations are correct. During our experiments, we discovered one bug in Alive2's constraint generation, which has been fixed by the developers.[3]

Benchmarks. We use as benchmarks five large programs used in the initial evaluation of Alive2 [25]: `bzip2`, `gzip`, `oggenc`, `ph7`, and `SQLite3`. We add to this set Fiat Cryptography [8], which contains verified C/C++ implementations of functions for cryptographic applications. We add this benchmark because it includes many straight-line math-intensive functions, a category that is largely missing from the other benchmarks. Fiat Cryptography's functions are also much larger (528 instructions on average, versus approximately 50), and they allow us to better investigate how spatial decomposition performs in both the straight-line and multi-basic block cases.

[3] https://github.com/AliveToolkit/alive2/issues/1290.

Table 2. Summary of LLVM IR functions in the evaluation dataset.

Benchmark	# Functions	# Instructions		
		Avg	Min	Max
`bzip2`	56	41.4	4	276
`oggenc`	213	49.2	5	587
`ph7`	857	47.9	5	1601
`sqlite3`	1446	50.0	4	5131
`gzip`	37	35.1	4	165
`fiat`	416	528.3	15	5681
Total	3025	114.8	4	5681

From these programs, we extract individual functions and test each in isolation, because Alive2 does not support interprocedural optimizations. The total number of functions across the benchmarks is 4,460, of which 3,116 are acyclic.[4] We excluded 48 functions that consist of two or fewer instructions, and 43 that are not supported by the existing Alive2 tool for various other reasons (*e.g.*, inline assembly). This process yielded 3,025 substantive functions, which we used to evaluate DESTIVA. Table 2 gives details about the benchmark set.

Experimental Setup. We conducted our experiments on a server with 96 CPUs and 512 GB RAM, running Ubuntu 24.04. We have used the latest development version of Alive2 available as of March 2026, Z3 version 4.12.3, and LLVM version 22.1. We limit each run of Alive2 and DESTIVA to 4 GB of memory and impose a 600-second timeout for each function. Unless stated otherwise, we report speedups as a geometric mean, and count Alive2 and DESTIVA timeouts as 600-second contributions. Alive2's pass-by-pass (P–P) method is, on average, more effective than end-to-end (E–E) translation validation for our benchmark set, with a performance difference of 1.562×. However, the outperformance is not uniform: on some benchmark functions, end-to-end is faster. In all of our experiments, therefore, we measure against both methods, running on the entire benchmark set, separately.

Research Questions. Our evaluation seeks to answer the following questions.

(1) **RQ1 (Spatial Decomposition)**: Can decomposing the input and output programs (the spatial axis) speed up translation validation?
(2) **RQ2 (Temporal Decomposition)**: Does dynamic grouping of transformations along the temporal axis speed up translation validation?
(3) **RQ3 (Combined Spatial and Temporal Decomposition)**: Can spatial and temporal decomposition be combined to yield further performance improvements?

5.1 RQ1: Spatial Decomposition

We evaluate the performance of DESTIVA along the spatial axis by comparing the performance of translation validation using spatial decomposition to the two existing translation validation methods used by Alive2; the results are shown in Table 3. For end-to-end translation validation, DESTIVA substantially outperforms Alive2, validating the translation of 271 functions for which Alive2 timed out, and speeding up solving by 1.179×. DESTIVA is even more effective at improving performance for functions whose translation is difficult to solve; for those that take Alive2 more

[4]In its original form, the Fiat Cryptography benchmark set contains 107 duplicate functions, which we remove during pre-processing.

Table 3. Speedup results for spatial decomposition, comparing DESTIVA to the two existing methods of Alive2. The "> 10s" columns give the results for benchmarks whose initial validation time exceeded 10 seconds.

(a) Spatial decomposition in the end-to-end translation validation context.

Timeout Cases	1445	
Timeout → Solved	271 (18.8%)	
	All	> 10s
Cases	3025	1987
Speedup	1.179x	1.960x
Portfolio Speedup	1.919x	2.335x

(b) Spatial decomposition applied to pass-by-pass translation validation.

Timeout Cases	1437	
Timeout → Solved	172 (12.0%)	
	All	> 10s
Cases	3025	1700
Speedup	0.676x	1.251x
Portfolio Speedup	1.273x	1.452x

than 10 seconds, spatial decomposition achieves a speedup of almost 2×. Spatial decomposition is also able to improve the performance of difficult benchmarks for pass-by-pass translation validation, as shown in Table 3b: DESTIVA validates the translation of 172 functions for which Alive2 times out, and speeds up problems that take more than 10 seconds by 1.251×.

DESTIVA is less effective for easier problems in the pass-by-pass case, actually slowing down solving relative to Alive2 over all cases. This slowdown occurs because the analysis required to construct f and f' incurs some overhead: for easy validation problems, this overhead makes up a larger proportion of the total runtime of DESTIVA, while for harder problems, SMT solving time dominates and DESTIVA achieves a large speedup.[5] Pass-by-pass translation validation (Table 3b) incurs the overhead of constructing f and f' once for every pass (*i.e.*, as many as 100 but typically 10-20 times for each function), which drives down performance on benchmarks with low initial validation time. For larger programs, however, our approach achieves significant speedups over Alive2. Figure 7 shows the geometric-mean speedup achieved for benchmarks that take more than x-many seconds to solve as x increases. DESTIVA achieves an average speedup at all initial running times in the end-to-end translation validation case, peaking just under 2× (Figure 7a). In the pass-by-pass case (Figure 7b), DESTIVA achieves a mean speedup for benchmarks with initial solving time above 2 seconds, peaking at approximately 1.3× speedup.

In both Table 3 and Figure 7, we show a portfolio result [46] in addition to the overall result; the portfolio numbers represent the speedup that can be achieved by running both DESTIVA and Alive2 in parallel and taking whichever result is produced first—a trade-off of computational resources for better performance. Even for functions with low original validation time in the pass-by-pass case, DESTIVA is able to achieve significant speedups under the portfolio methodology, when the original Alive2 result can be used to offset cases where overhead is significant. Table 4 shows the results broken down by benchmark. Spatial decomposition is most effective for `gzip` and least effective for `fiat`, although at least some speedup is achieved for all benchmarks with initial validation time exceeding 10 seconds. There is some variation among benchmarks between end-to-end and pass-by-pass; `ph7` is sped up dramatically using end-to-end validation, but very little with pass-by-pass validation.

To better understand where the benefits of spatial decomposition accrue, we also measure the speedups achieved for each LLVM optimization pass; the results are shown in Table 5. The "Functions" column gives the number of functions from the benchmark set where the pass made a substantive change, and "Invocations" gives the total number of times the pass made a substantive

[5]It is hard to directly measure the overhead added by DESTIVA due to its integration with Alive2, so we measure the effect by proxy in Figure 7.

Table 4. Breakdown of spatial decomposition results by benchmark.

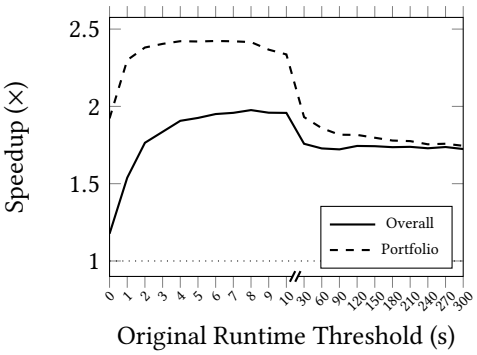
	End-to-end				Pass-by-pass			
	Overall	>10s	Portfolio	T→S	Overall	>10s	Portfolio	T→S
bzip2	1.244×	2.063×	1.709×	8 (38%)	0.665×	1.182×	1.307×	3 (14%)
fiat	0.806×	1.035×	1.446×	91 (37%)	0.424×	1.118×	1.163×	63 (26%)
gzip	1.460×	2.217×	1.641×	5 (31%)	1.672×	3.772×	2.073×	5 (33%)
oggenc	1.365×	2.213×	1.894×	19 (22%)	0.657×	1.682×	1.437×	13 (15%)
ph7	2.168×	3.494×	3.196×	48 (12%)	0.776×	1.175×	1.233×	24 (6%)
sqlite3	0.891×	1.621×	1.554×	100 (15%)	0.700×	1.274×	1.289×	64 (10%)
Overall	1.179×	1.960×	1.919×	271 (19%)	0.676×	1.251×	1.273×	172 (12%)

Table 5. Speedups achieved and proportion of instructions included in spatial decomposition for each LLVM optimization pass under pass-by-pass methodology.

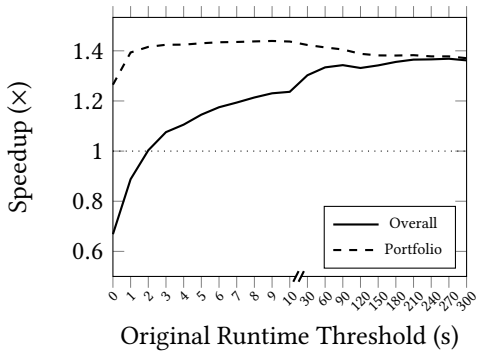
Pass	Functions	Invocations	$\lvert f\rvert/\lvert P\rvert$	Speedup
SROA	2910	2912	87.8%	1.44×
InstCombine	1220	1303	71.1%	2.09×
SimplifyCFG	900	989	18.3%	13.51×
EarlyCSE	725	734	63.9%	3.55×
CorrelatedValuePropagation	144	147	39.3%	6.57×
Reassociate	129	130	28.6%	15.38×
SLPVectorizer	32	32	98.7%	7.91×
MemCpyOpt	31	32	68.4%	0.14×
Others (15 passes)	68	89	26%–66%	0.55×–6.78×

change. The -O3 optimization pipeline invokes some passes, like InstCombine, more than once per function. For each pass, we report the geometric-mean speedup achieved, and the proportion of P's instructions included in f, to measure how many instructions are being removed during spatial decomposition. During our experiments, a total of 23 passes made some change to a function, but of these, 15 affected less than 1% of the benchmark functions, which we group under "Others" in Table 5. The greatest speedups are achieved for the SimplifyCFG and Reassociate passes, while MemCpyOpt is slowed down (though it affects only 31 functions). The results also show that larger speedups correspond to those passes where spatial decomposition eliminates the most instructions: the size of f is smallest for the two passes sped up the most. The two passes with most frequent use were SROA and InstCombine, for which spatial decomposition trimmed away 12% and 29% of instructions, respectively, and achieved speedups of 1.44× and 2.09×.

Findings. Spatial decomposition achieves speedups compared to both end-to-end (1.96×) and pass-by-pass (1.25×) translation validation. However, speedups are only significant for benchmarks that take a long time to solve; for easier benchmarks, analysis overhead may exceed the benefit and slow down validation using the pass-by-pass method. Spatial decomposition is most effective for gzip, but speeds up each benchmark subset for functions that exceed 10 seconds of initial validation time. Spatial decomposition sped up SimplifyCFG and Reassociate the most, but the biggest impact was achieved on SROA and InstCombine, which affect more functions.



(a) Speedup versus initial running-time threshold compared to end-to-end translation validation.



(b) Speedup versus initial running-time threshold compared to pass-by-pass translation validation.

Fig. 7. Graphs of speedup versus initial validation time threshold for spatial decomposition. An initial threshold of x means that functions that take fewer than x seconds are excluded from the computation.

5.2 RQ2: Temporal Decomposition

Along the temporal axis, we evaluate each of the temporal decomposition heuristics described in Section 4.4, comparing each against both the pass-by-pass (P–P) and end-to-end (E–E) existing modes of Alive2. The results are given in Table 6. All of the temporal decomposition heuristics achieve substantial speedups over the existing end-to-end (E–E) method, with speedups ranging from 1.5× to 1.9×; following portfolio methodology, the speedups are even greater. Temporal decomposition is less effective compared to the existing pass-by-pass approach, and two heuristics (Cutout, and Top2) degrade performance. The Fixed heuristic performs surprisingly well, speeding up translation validation by 1.23× compared to the pass-by-pass method and approximately 2× compared to the end-to-end method. The most effective heuristics, though, are Clustered and Clustered4, which achieve speedups of 2.7× compared to end-to-end and 1.9× compared to pass-by-pass. The Cutout and Top2 heuristics have worse performance than pass-by-pass validation, though they do outperform end-to-end translation validation, where at least some level of decomposition appears to help. All five tested heuristics divided the transformation sequence into a similar number of segments (between 3.5 and 5), but the performance results vary substantially. This result suggests that the choice of which passes to group together makes a substantial difference in performance.

Findings. Overall, our results show that temporal decomposition speeds up translation validation substantially: up to 2.7× for difficult benchmark functions relative to end-to-end translation validation. The significant variation in effectiveness of temporal decomposition heuristics suggests that pass grouping affects performance; the most effective heuristic we tested was Clustered4.

5.3 RQ3: Combined Spatial and Temporal Decomposition

Table 7 shows the results when spatial and temporal decomposition are combined; the difference from Table 6 shows the advance made by the combination. Against end-to-end validation, adding spatial decomposition does not substantially increase average speedups, but it validates many more functions that timed out under temporal decomposition alone (381 versus 289 with Clustered4, for example). Against pass-by-pass validation, it makes significant gains, both in speedups for difficult functions and in the number of timeout cases validated. As in Section 5.1, adding spatial decomposition slows down easy validation tasks because overhead exceeds the benefit.

Table 6. Speedup results for temporal decomposition with various heuristics over the entire benchmark set (n = 3,025). The rows labeled "T → S" give the number of functions which timed out originally but are solved with the heuristic.

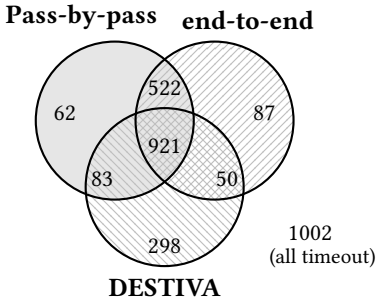
		Fixed	Cutout	Top2	Clus	Clus4	P–P	E–E
# of Segments (Average)		4.00	4.03	4.78	4.93	3.52	112.04	1
Versus E–E	T → S	216	133	137	208	289	145	–
	Overall	1.923×	1.548×	1.532×	1.647×	1.935×	1.562×	–
	>10s	2.409×	1.986×	1.990×	2.146×	2.721×	2.032×	–
	Portfolio	2.097×	1.833×	1.823×	1.886×	2.195×	1.845×	–
Versus P–P	T → S	140	10	15	88	184	–	137
	Overall	1.231×	0.991×	0.981×	1.281×	1.505×	–	0.640×
	>10s	1.258×	0.999×	0.992×	1.396×	1.855×	–	1.074×
	Portfolio	1.267×	1.048×	1.026×	1.333×	1.565×	–	1.181×

Table 7. Speedup results for combined spatial and temporal decomposition with various temporal heuristics. The rows labeled "T → S" give the number of functions which timed out originally but are solved with the heuristic.

		Fixed	Cutout	Top2	Clus	Clus4	P–P	E–E
# of Segments (Average)		4.00	4.03	4.78	4.93	3.52	112.04	1
Versus E–E	T → S	292	202	204	265	381	145	–
	Overall	1.219×	1.029×	1.050×	1.259×	1.577×	1.562×	–
	>10s	2.141×	1.822×	1.849×	2.016×	2.765×	2.032×	–
	Portfolio	2.017×	1.806×	1.822×	1.907×	2.357×	1.845×	–
Versus P–P	T → S	255	165	170	227	348	–	137
	Overall	0.778×	0.661×	0.680×	0.975×	1.225×	–	0.640×
	>10s	1.436×	1.223×	1.234×	1.673×	2.428×	–	1.074×
	Portfolio	1.391×	1.254×	1.263×	1.574×	1.943×	–	1.181×

The best-performing heuristic is again Clustered4; Figure 8b therefore reports results using this heuristic. DESTIVA achieves speedups over both existing modes of Alive2, and for difficult functions, the combination of spatial and temporal decomposition always outperforms either method on its own. Relative to end-to-end translation validation, the combination performs only slightly better than temporal decomposition alone, but against pass-by-pass validation, the combined speedup (2.43×) far exceeds spatial (1.25×) or temporal (1.86×) decomposition alone. The combined method also succeeds in validating nearly twice as many timeout cases as either spatial decomposition or temporal decomposition alone.

However, as Figure 8a shows, while DESTIVA solves 298 benchmarks on which both existing approaches fail, it fails on 584 benchmarks for which either pass-by-pass or end-to-end validation succeeds. These results indicate that, despite average speedups, spatial and temporal decomposition have high costs. Portfolio methodology offsets these costs but requires twice as many computational resources. In our experiments, portfolio methodology yields higher speedups relative to both existing approaches, and cases where DESTIVA times out can default to an existing method.



(a) A Venn diagram of the cases that can be handled by each approach.

		Spatial	Temporal	Combined
Versus E–E	T → S	207	289	381
	Overall	1.058×	1.935×	1.577×
	>10s	1.884×	2.721×	2.765×
	Portfolio	1.847×	2.195×	2.357×
Versus P–P	T → S	172	184	348
	Overall	0.676×	1.505×	1.225×
	>10s	1.251×	1.855×	2.428×
	Portfolio	1.273×	1.565×	1.943×

(b) Overall results for DESTIVA with spatial decomposition, temporal decomposition, and both combined.

Fig. 8. Results of running DESTIVA with both spatial and temporal decomposition.

Because timeouts also muddle the speedup measurements, we additionally measure speedups for the 921 benchmarks that all three approaches solve (the center of Figure 8a): here DESTIVA speeds up solving by 2.42× over end-to-end validation but only 1.07× over pass-by-pass validation. Overhead is again significant for small benchmarks: on the functions taking longer than 10 seconds (114/921), the speedup over pass-by-pass is 4.8×.

Findings. The combination of temporal and spatial decomposition outperforms either mode on its own, speeding up translation validation by up to 2.7× relative to end-to-end and 2.4× relative to pass-by-pass validation. However, DESTIVA times out in many cases that one or both existing approaches solve, as shown in Figure 8a, so portfolio methodology may be required in practice.

6 Discussion and Future Work

Reliance on Compiler Information. In this work, we rely on information extracted from LLVM via instrumentation (*i.e.*, the tags described in Section 3.3) to speed up translation validation of the LLVM compiler—this is potentially circular reasoning. For temporal decomposition, compiler information does not affect soundness: just as with the existing strategy of pass-by-pass translation validation, Theorem 4.9 guarantees correctness no matter what cut points are chosen. For spatial decomposition, we describe in Section 4.3 the circumstances in which a compiler bug could affect soundness; our approach is sound under the condition that the compiler information is correct. A bug that adds *too many* instructions to A or A' may reduce the *performance* benefit of decomposition, but it does not harm *correctness*. The only way for a compiler bug to lead to an unsound result is for the compiler to erroneously strip custom debug information tags from the program under optimization. While we therefore cannot prove the absence of such bugs, we implement DESTIVA to make them unlikely in practice, and found no unsound results during our extensive evaluation.

Portfolio Methodology. The results in Section 5 suggest that portfolio methodology is key to reducing the number of cases where DESTIVA times out; we found 584 such cases during our experiments. Portfolio methodology has been used extensively for SMT and SAT solving [29, 46, 47], and may benefit translation validation as well. Nor is it limited to running DESTIVA alongside an existing validation mode: different configurations of spatial and temporal decomposition could themselves form a portfolio. In our experiments, 461 functions that time out with only spatial decomposition are solved with only temporal decomposition, and 456 the reverse, making the combination of different DESTIVA configurations promising future work.

Support for Loops. DESTIVA does not directly support programs that contain loops: the ordering argument in the proof of Lemma 4.4 does not hold for control-flow graphs with cycles. There are two primary existing methods for handling loops during translation validation, and our approach is compatible with either. The first is simple loop unrolling, the approach taken by Alive2 [25]: loops are unrolled up to a user-specified bound n, producing an acyclic control-flow graph to which spatial decomposition applies directly, inheriting the same limitations as plain solver-based reasoning. The second is program alignment [5], which pairs up iterations of basic blocks between P and P' and proves their equivalence for any number of iterations; spatial decomposition could be applied within aligned blocks, eliminating unchanged instructions just as for acyclic programs. The choice of loop-handling strategy is largely orthogonal to our approach—our temporal decomposition heuristics are compatible with any of them—so we leave this integration to future work.

Portability. Our compiler instrumentation and spatial decomposition are implemented on LLVM's intermediate representation, but neither is conceptually tied to it: temporal decomposition applies wherever a sequence of program transformations is carried out, and spatial decomposition to any language with instructions and a control flow graph. For higher-level languages, the proofs of Section 4 would need only slight adaptation to handle data flow dependencies outside SSA form. For low-level SSA languages such as GCC's intermediate representation, porting DESTIVA would require minimal changes to the debug information tagging and could build on the existing SMT-GCC [43] translation validation tool.

Decomposition Granularity. Our temporal decomposition heuristics operate at the granularity of passes: they group passes together but treat each pass as an indivisible atom of transformation. Future work could decompose *within* a single pass—for instance, between the iterations of passes that internally run to a fixed point—and explore richer compiler instrumentation to guide such intra-pass decomposition.

7 Related Work

Translation Validation. Our work is related to prior literature on translation validation and the broader category of equivalence checking. Translation validation was introduced by Pnueli et al. [33]; Necula [32] applied it in practice to an optimizing compiler. Since these early works, many improvements on the basic translation validation strategy have been developed; these have largely focused on two methods of performing translation validation. One approach, *normalizing translation validation* [41], is to rewrite the expressions of P' and/or P until the two programs are syntactically equal [12, 23, 24, 37, 40, 41]. The second approach is to use a solver or other symbolic reasoning tool to compare the semantics of P and P' [2, 5, 9, 13, 14, 22, 35, 49]; this is the approach of Necula [32], which describes it as symbolic execution. Alive and Alive2 [3, 25–27] are in the second category: they generate SMT constraints and call Z3 [7]. Walfridsson [43] takes the same approach for GCC as Alive2 does for LLVM. Our approach differs from both methods in that it attempts to simplify the problem above the level of either constraint-based or syntax-based reasoning; conceptually, both spatial and temporal decomposition could be applied to either. Existing approaches to translation validation have also proposed cut points as a strategy for translation validation [11, 48]. Cut points are program locations where the relationship between P and P' is known; they therefore allow the task of translation validation to be carried out for each segment between cut points. While this approach does decompose programs along what we call the spatial axis, it only breaks them into smaller pieces and never removes any instructions from consideration by an underlying solver. Our spatial decomposition strategy takes a different tack, entirely removing irrelevant sections of a program during the production of f and f'.

Program and Transformation Decomposition. Our work on spatial and temporal decomposition for translation validation is also related to prior work on decomposing both programs and transformation sequences. Wang et al. [45] introduce a method for finding minimal constructs in a hardware design required to prove that two designs are equivalent; this approach is conceptually similar to our spatial decomposition, but relies on particular characteristics of hardware designs, rather than programs. SYMDIFF [23] uses an intermediate verification language to find differences in the behavior of two programs and identify which program statements were associated with semantics changes; we perform decomposition in the opposite order, using the results of compiler instrumentation to reduce the verification burden. Our approach also bears some similarity to work that decomposes large codebases for automatic translation [16, 17, 44]; however, we perform decomposition at a lower granularity and for the purpose of translation validation for an optimizing compiler, rather than code translation at the level of source code. Our temporal decomposition approach is related to prior work on equivalence checking for versions of the same program [10, 38]; Badihi et al. [1] even extract common code from versions of a single program, like our spatial decomposition strategy. Our method differs from this line of work because we aim to improve translation validation performance rather than to check the equivalence of human- or LLM-written versions of source code programs. Milovančević and Kunčak [30] propose clustering similar programs from a large dataset by similarity, thereby reducing validation effort between each cluster. This approach differs from our temporal decomposition strategy in that we use a linear sequence of optimizations instead of clustering multiple program versions.

Compiler Instrumentation. Our technique uses compiler instrumentation to detect which instructions are modified during a run of LLVM's optimizer. Several existing translation validation tools also make use of compiler instrumentation to guide their validation [19, 21, 31, 32], but to our knowledge, our approach is the first to use the extracted compiler information to guide decomposition, either of the input program or of the transformation pass sequence. Outside the realm of translation validation, Huang et al. [15] also instrument LLVM optimization passes with the goal of finding bugs in debug information updates. Chaliasos et al. [4] instrument compilers for higher-level languages to measure coverage of compiler components during a testing campaign. Our approach differs in that we aim to extract only the simple set of instructions modified during an optimization, and in that the goal of our compiler instrumentation is to obtain information that can be used for spatial and temporal decomposition.

8 Conclusion

This paper has presented a novel method for speeding up translation validation by decomposing the monolithic translation validation problem across both program contents (the spatial axis) and a sequence of optimizations (the temporal axis). Our primary contribution is along the spatial axis, where we instrument the compiler to record which instructions are changed by an optimization, and then perform a program chop based on two-way data flows to extract a program fragment whose validation result can be soundly extrapolated to the whole program. Along the temporal axis, we introduce several new heuristics that harness the results of compiler instrumentation to strike a balance between validating the entire transformation sequence in one go and validating each optimization step on its own. Our experiments found that spatial decomposition improves performance relative to a state-of-the-art tool, and that temporal decomposition contributes a further improvement when the two are combined. Future work could include improving DESTIVA's performance using portfolio methodology, and extending it to support loops. Future work could also extend our approach to another compiler, such as GCC, and investigate whether temporal decomposition is possible within individual compiler passes.

Data-Availability Statement

The implementation of DESTIVA is as described in Section 5: it consists of instrumentation of LLVM's low-level APIs (Section 3.3) and significant modifications to the existing tool Alive2 to support both temporal and spatial decomposition. The implementation of DESTIVA and our experimental results have been persistently archived [28]. The latest version of DESTIVA is also made publicly available on GitHub.[6]

Acknowledgments

We thank the anonymous reviewers of our work for feedback on earlier drafts of this paper. The work described in this paper was supported, in part, by the United States National Science Foundation (NSF) under grants No. 2114627 and No. 2237440. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

References

[1] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/3368089.3409757

[2] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. 2005. TVOC: A Translation Validator for Optimizing Compilers. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–295. doi:10.1007/11513988_29

[3] Ryan Berger, Mitch Briles, Nader Boushehrinejad Moradi, Nicholas Coughlin, Kait Lam, Nuno P. Lopes, Stefan Mada, Tanmay Tirpankar, and John Regehr. 2025. Translation Validation for LLVM's AArch64 Backend. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 369 (Oct. 2025), 26 pages. doi:10.1145/3763147

[4] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 183–198. doi:10.1145/3519939.3523427

[5] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1027–1040. doi:10.1145/3314221.3314596

[6] Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. 2017. Predicting rankings of software verification tools. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics* (Paderborn, Germany) *(SWAN 2017)*. Association for Computing Machinery, New York, NY, USA, 23–26. doi:10.1145/3121257.3121262

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[8] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1202–1219. doi:10.1109/SP.2019.00005

[9] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property Directed Equivalence via Abstract Simulation. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 433–453. doi:10.1007/978-3-319-41540-6_24

[10] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 349–360. doi:10.1145/2642937.2642987

[11] Xiushan Feng and Alan J. Hu. 2005. Cutpoints for formal equivalence verification of embedded software. In *Proceedings of the 5th ACM International Conference on Embedded Software* (Jersey City, NJ, USA) *(EMSOFT '05)*. Association for Computing Machinery, New York, NY, USA, 307–316. doi:10.1145/1086228.1086284

[6] https://github.com/mikekben/destiva

[12] Carsten Fuhs, Cynthia Kop, and Naoki Nishida. 2017. Verifying Procedural Programs via Constrained Rewriting Induction. *ACM Trans. Comput. Logic* 18, 2, Article 14 (June 2017), 50 pages. doi:10.1145/3060143

[13] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 221 (Nov. 2020), 29 pages. doi:10.1145/3428289

[14] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards modularly comparing programs using automated theorem provers. In *Proceedings of the 24th International Conference on Automated Deduction* (Lake Placid, NY) *(CADE'13)*. Springer-Verlag, Berlin, Heidelberg, 282–299. doi:10.1007/978-3-642-38574-2_20

[15] Shan Huang, Jingjing Liang, Ting Su, and Qirun Zhang. 2025. Robustifying Debug Information Updates in LLVM via Control-Flow Conformance Analysis. *Proc. ACM Program. Lang.* 9, PLDI, Article 168 (June 2025), 23 pages. doi:10.1145/3729267

[16] Ali Reza Ibrahimzada. 2024. Program Decomposition and Translation with Static Analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 453–455. doi:10.1145/3639478.3641226

[17] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE109 (June 2025), 23 pages. doi:10.1145/3729379

[18] Daniel Jackson and Eugene J. Rollins. 1994. A new model of program dependences for reverse engineering. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering* (New Orleans, Louisiana, USA) *(SIGSOFT '94)*. Association for Computing Machinery, New York, NY, USA, 2–10. doi:10.1145/193173.195281

[19] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. 2006. A PVS Based Framework for Validating Compiler Optimizations. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM '06)*. IEEE Computer Society, USA, 108–117. doi:10.1109/SEFM.2006.4

[20] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. 2009. Validation of GCC optimizers through trace generation. *Softw. Pract. Exper.* 39, 6 (April 2009), 611–639. doi:10.1002/spe.913

[21] 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. 2018. Crellvm: verified credible compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 631–645. doi:10.1145/3192366.3192377

[22] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1004–1019. doi:10.1145/3445814.3446751

[23] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 712–717. doi:10.1007/978-3-642-31424-7_54

[24] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 345–355. doi:10.1145/2491411.2491452

[25] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. doi:10.1145/3453483.3454030

[26] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 22–32. doi:10.1145/2737924.2737965

[27] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 317–337. doi:10.1007/978-3-662-53413-7_16

[28] Benjamin Mikek, Chathur Bommineni, Qirun Zhang, and Thomas Reps. 2026. *DESTIVA: DEcomposing Spatially and Temporally for translatIon VAlidation.* doi:10.5281/zenodo.22716231

[29] Benjamin Mikek and Qirun Zhang. 2023. Speeding up SMT Solving via Compiler Optimization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1177–1189. doi:10.1145/3611643.3616357

[30] Dragana Milovančević and Viktor Kunčak. 2023. Proving and Disproving Equivalence of Functional Programming Assignments. *Proc. ACM Program. Lang.* 7, PLDI, Article 144 (June 2023), 24 pages. doi:10.1145/3591258

[31] Kedar S. Namjoshi and Lenore D. Zuck. 2013. Witnessing Program Transformations. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–323. doi:10.1007/978-3-642-38856-9_17

[32] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. doi:10.1145/349299.349314

[33] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166. doi:10.1007/bfb0054170

[34] Thomas W. Reps and Genevieve Rosay. 1995. Precise Interprocedural Chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1995, Washington, DC, USA, October 10-13, 1995*, Gail E. Kaiser (Ed.). ACM, 41–52. doi:10.1145/222124.222138

[35] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 471–482. doi:10.1145/2491956.2462183

[36] C. Spearman. 1904. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 15, 1 (1904), 72–101. http://www.jstor.org/stable/1412159

[37] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based translation validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 737–742. doi:10.1007/978-3-642-22110-1_59

[38] Ofer Strichman and Benny Godlin. 2005. *Regression Verification - A Practical Way to Verify Programs*. Springer-Verlag, Berlin, Heidelberg, 496–501. doi:10.1007/978-3-540-69149-5_54

[39] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2024. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. In *Software Engineering 2024, Fachtagung des GI-Fachbereichs Softwaretechnik, Linz, Austria, February 26 - March 1, 2024 (LNI, Vol. P-343)*, Rick Rabiser, Manuel Wimmer, Iris Groher, Andreas Wortmann, and Bianca Wiesmayr (Eds.). Gesellschaft für Informatik e.V., 67–68. doi:10.18420/sw2024_18

[40] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. doi:10.1145/1480881.1480915

[41] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 295–305. doi:10.1145/1993498.1993533

[42] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 17–27. doi:10.1145/1328438.1328444

[43] Krister Walfridsson. [n. d.]. smtgcc: Some Experiments with SMT Solvers and GIMPLE IR. Accessed: 2026-02-10. https://github.com/kristerw/smtgcc

[44] Bo Wang, Tianyu Li, Ruishi Li, Umang Mathur, and Prateek Saxena. 2025. Program Skeletons for Automated Program Translation. *Proc. ACM Program. Lang.* 9, PLDI, Article 184 (June 2025), 25 pages. doi:10.1145/3729287

[45] Yanzhao Wang, Fei Xie, Zhenkun Yang, Pasquale Cocchini, and Jin Yang. 2023. An Equivalence Checking Framework for Agile Hardware Design. In *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 26–32. doi:10.1145/3566097.3567843

[46] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. 2009. A Concurrent Portfolio Approach to SMT Solving. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*. Springer, 715–720. doi:10.1007/978-3-642-02658-4_60

[47] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *J. Artif. Int. Res.* 32, 1 (June 2008), 565–606. doi:10.1613/jair.2490

[48] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Proceedings of the 15th International Symposium on Formal Methods* (Turku, Finland) *(FM '08)*. Springer-Verlag, Berlin, Heidelberg, 35–51. doi:10.1007/978-3-540-68237-0_5

[49] Lenore D. Zuck, Amir Pnueli, Benjamin Goldberg, Clark W. Barrett, Yi Fang, and Ying Hu. 2005. Translation and Run-Time Validation of Loop Transformations. *Formal Methods Syst. Des.* 27, 3 (2005), 335–360. doi:10.1007/s10703-005-3402-z