Proving Optimizations Correct using Parameterized Program Equivalence *

Sudipta Kundu Zachary Tatlock Sorin Lerner
University of California, San Diego
{skundu,ztatlock,lerner}@cs.ucsd.edu

Abstract

Translation validation is a technique for checking that, after an optimization has run, the input and output of the optimization are equivalent. Traditionally, translation validation has been used to prove concrete, fully specified programs equivalent. In this paper we present Parameterized Equivalence Checking (PEC), a generalization of translation validation that can prove the equivalence of parameterized programs. A parameterized program is a partially specified program that can represent multiple concrete programs. For example, a parameterized program may contain a section of code whose only known property is that it does not modify certain variables. By proving parameterized programs equivalent, PEC can prove the correctness of transformation rules that represent complex optimizations once and for all, before they are ever run. We implemented our PEC technique in a tool that can establish the equivalence of two parameterized programs. To highlight the power of PEC, we designed a language for implementing complex optimizations using many-to-many rewrite rules, and used this language to implement a variety of optimizations including software pipelining, loop unrolling, loop unswitching, loop interchange, and loop fusion. Finally, to demonstrate the effectiveness of PEC, we used our PEC implementation to verify that all the optimizations we implemented in our language preserve program behavior.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – *correctness proofs, reliability, validation*; D.3.4 [*Programming Languages*]: Processors – *compilers, optimization*

General Terms Reliability, Languages, Verification.

Keywords Compiler Optimization, Correctness, Translation Validation.

1. Introduction

Compilers are a fundamental component of the toolset programmers rely on every day. As a result, compiler correctness is crucially important. A bug in the compiler can systematically introduce errors in each generated executable. Furthermore, compiler bugs can invalidate strong guarantees that were established on the original source program. As an example, various analysis tools can prove the absence of certain kinds errors at the source level (e.g. double locking or null pointer exceptions). However, if the compiler is not guaranteed to be correct, then no source-level guarantees can be safely transferred to the generated code. Finally, compiler correctness is all the more important in high-assurance domains like avionics and medical equipment, where the cost of incorrect compilation can be extremely high.

Unfortunately, building reliable compilers is difficult, error-prone, and requires significant manual effort. Indeed, it takes a long time to develop a compiler that is stable enough for broad adoption (often up to a decade), which in turn hinders the development of new languages and architectures.

One of the most error prone parts of a compiler is its optimization phase. Many optimizations require an intricate sequence of complex transformations. Often these transformations interact in unexpected ways, leading to a combinatorial explosion in the number of cases that must be considered to ensure that the optimization phase is correct. In fact, even mature compilers have optimization bugs and as a result professional developers sometimes go as far as disabling optimizations in critical modules to lower the likelihood of incorrectly generated code. Unfortunately, it is becoming less and less feasible to increase the reliability of a compiler by simply disabling most of its optimizations: with the widespread adoption of systems whose good performance depends heavily on compiler optimizations – for example just-in-time compilers and higher-level languages – turning off the optimizer is no longer an option. Furthermore, the difficulties of developing correct optimizations also prevent end-user programmers (non-compiler experts) from extending the compiler with simple custom optimizations, making most compilers closed black boxes, rather than open-ended extensible frameworks.

Previous techniques for providing correctness guarantees for optimizations can be divided into two categories. In the first category optimizations are proved correct *once and for all* [7, 32, 8, 2, 14, 15, 16]. In this setting, to prove that an optimization is correct, one must prove that for any input program the optimization produces a semantically equivalent program. The second category consists of proving correctness *each time an optimization is run*. Here, each time the compiler runs an optimization, an automated tool tries to prove that the original program and the corresponding optimized program are equivalent. This technique, which is called translation validation, has been successfully applied in a variety of settings, including mature optimizing compilers [20, 19, 22, 6, 33, 28, 29, 24], refinement checking of CSP programs [11], and high-level synthesis validation [12].

The primary advantage of once-and-for-all techniques is that they provide a very strong guarantee: optimizations are known to be correct when the compiler is built, before they are run even once.

* Supported in part by NSF grants CCF-0644306 and CCF-0811512.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

In contrast, translation validation provides a weaker correctness guarantee. This is because translation validation guarantees that only a particular run of the optimization is correct. Compilers that include translation validation may still contain bugs and it is unclear what a programmer should do when the compiler is unable to correctly compile a program.

On the other hand, translation validation has a clear advantage over once-and-for-all techniques in terms of automation. Most of the techniques that provide once-and-for-all guarantees require user interaction. Those that are fully automated, for example Cobalt [14] and Rhodium [15], work by having programmers implement optimizations in a domain-specific language using flow functions and single-statement rewrite rules. Unfortunately, the set of optimizations that these techniques can prove correct has lagged behind translation validation. In particular, translation validation can already handle complex loop optimizations like skewing, splitting and interchange, which have thus far eluded automated once-and-for-all approaches. A common intuition is that once-and-for-all proofs are harder to achieve because they must show that any application of the optimization is correct, as opposed to a single instance.

In this paper, we present a new technique for proving optimizations correct called Parameterized Equivalence Checking (PEC) that bridges the gap between translation validation and once-and-for-all techniques. PEC generalizes translation validation to handle parameterized programs, which are partially specified programs that can represent multiple concrete programs. For example, a parameterized program may contain a section of code whose only known property is that it does not define or use a particular variable.

The key insight of PEC is that existing translation validation techniques can be adapted to work in the broader setting of parameterized programs. This allows translation validation techniques, which have traditionally been used to prove *concrete* programs equivalent, to prove *parameterized* programs equivalent. Most importantly, because optimizations can be expressed as nothing more than parameterized transformation rules, using before and after parameterized code patterns, PEC can prove once and for all that such optimizations preserve semantics.

To highlight the power and generality of PEC, we designed a new language for writing optimizations, and implemented a checker based on PEC that can automatically check the correctness of optimizations written in this language. Our language for implementing and proving optimizations correct is much more expressive than previous such optimization languages, like Cobalt [14] and Rhodium [15]: whereas Cobalt and Rhodium only supported local rewrites of a single statement to another, our language supports many-to-many rewrite rules. Such rules are able to replace an entire set of statements, even entire loops and branches, with a completely different set of statements. Using these rules, we can express many more optimizations than in Cobalt and Rhodium, and we can also prove them all correct using our PEC algorithm.

In summary, our contributions are:

- We developed and implemented a technique for performing Parameterized Equivalence Checking. PEC adapts two approaches from traditional translation validation to the setting of once-and-for-all correctness proofs, namely the relational approach of Necula [19], and the permute approach of Zuck *et al.* [33].
- We developed a new language for implementing optimizations. Our language is more expressive than previous languages that can be checked for correctness automatically: it has explicit support for expressing many-to-many transformations, meaning that a set of statements can be transformed to another set of statements in a single rewrite.

```
i := 0
while (i < n) {
   a[i] += 1;
   b[i] += a[i];
   c[i] += b[i];
   i++;
}
```

(a)

```
a[0] += 1;
b[0] += a[0];
a[1] += 1;
i := 0
while (i < n - 2) {
   a[i+2] += 1;
   b[i+1] += a[i+1];
   c[i] += b[i];
   i++
}
c[i] += b[i];
b[i+1] += a[i+1];
c[i+1] += b[i+1];
```

(b)

Figure 1. Software pipelining: (a) shows the original code, and (b) shows the optimized code.

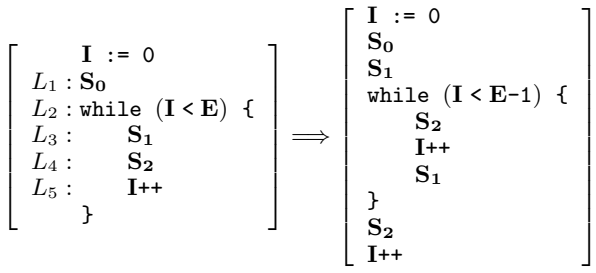
- We implemented and proved correct a variety of complex optimizations in our system. Some of these optimizations, for example partial redundancy elimination, could have been expressed and proved correct in Rhodium [15], but they are easier to express in our language because of the built-in support for many-to-many rewrite rules. Furthermore, many of the optimizations we implemented and proved correct could not be proved correct or even expressed in previous systems like Rhodium. This includes: software pipelining, loop unswitching, loop unrolling, loop peeling, loop splitting, loop alignment, loop interchange, loop skewing, loop reversal, loop fusion and loop distribution.

The rest of the paper is organized as follows. Section 2 presents an overview of our approach through an example. Section 3 describes our PEC system at an architectural level, identifying its three main modules. The following three sections (Sections 4, 5 and 6) present each of the three modules in more detail. Finally, Section 7 presents our experimental results.

2. Overview

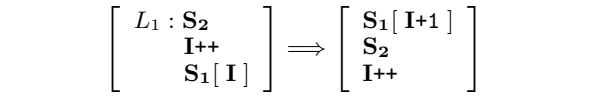
We illustrate the main ideas of our approach through an example: software pipelining. Software pipelining is an optimization that tries to break dependencies within a loop body by spreading instructions from one iteration in the original program across multiple iterations of the transformed program. Software pipelining can break dependencies inside a loop body, and thus provides more flexibility to the scheduler, but it does so without increasing the code size of the loop body (as opposed to loop unrolling, which also provides more flexibility to the scheduler, but may have adverse effects on the cache by increasing the loop body size).

As an example, consider the code in Figure 1(a). The loop updates three arrays iteratively, but because each update depends on the previous one, each instruction in the loop must wait until the instruction immediately before it finishes. Figure 1(b) shows the result of applying software pipelining on this loop. The key insight is that, in the steady state, the transformed loop still runs the same three instructions from the original loop, but now each of the three instructions is from a different iteration of the original loop. The `a[i+2]` instruction runs two iterations ahead; the `b[i+1]` runs one iteration ahead; and the `c[i]` instruction runs on the current iteration. In order to get into this steady state, one has to add a prologue at the beginning of the transformed loop in order to setup the pipelining effect. There is also an epilogue after the loop to execute the remaining instructions.



where
DoesNotModify$(\mathbf{S_0}, \mathbf{I})@L_1 \wedge$
DoesNotModify$(\mathbf{S_1}, \mathbf{I})@L_3 \wedge$ *DoesNotModify*$(\mathbf{S_2}, \mathbf{I})@L_4 \wedge$
StrictlyPositive$(\mathbf{E})@L_2 \wedge$ *DoesNotModify*$(\mathbf{S_1}, \mathbf{E})@L_3 \wedge$
DoesNotModify$(\mathbf{S_2}, \mathbf{E})@L_4 \wedge$ *DoesNotModify*$(\mathbf{I{+}{+}}, \mathbf{E})@L_5$

Figure 2. First part of software pipelining



where *DoesNotModify*$(\mathbf{S_2}, \mathbf{I})@L_1 \wedge$
Commute$(\mathbf{S_2}, \mathbf{S_1}[\ \mathbf{I{+}1}\ ])@L_1$

Figure 3. Second part of software pipelining

fact *StrictlyPositive*$(\mathbf{E})$
has meaning $eval(\sigma, \mathbf{E}) > 0$

fact *DoesNotModify*$(\mathbf{S}, \mathbf{E})$
has meaning $eval(\sigma, \mathbf{E}) = eval(step(\sigma, \mathbf{S}), \mathbf{E})$

fact *Commute*$(\mathbf{S_1}, \mathbf{S_2})$
has meaning $step(step(\sigma, \mathbf{S_1}), \mathbf{S_2}) = step(step(\sigma, \mathbf{S_2}), \mathbf{S_1})$

Figure 4. Meanings of some facts that we use in our system

Software pipelining is a non-trivial optimization, with many subtle corner cases that need to be correctly implemented. The prologue and the epilogue must execute a precisely crafted sequence of instructions to setup and unwind the steady state; the instructions in the loop must be correctly re-indexed; and the entire optimization must be applied only if no dependencies in the original program would be broken by software pipelining.

2.1 Expressing Software Pipelining

We implement software pipelining in our language as the repeated application of two simple optimizations. In Figure 2 we show the first one, which simply moves some instructions (namely $\mathbf{S_1}$) from the current iteration to the next iteration. Optimizations in our language are written as parameterized rewrite rules with side conditions: $P_1 \Longrightarrow P_2$ **where** ϕ, where P_1 and P_2 are parameterized programs, and ϕ is a side condition that states when the rewrite rule can safely be fired. An optimization $P_1 \Longrightarrow P_2$ **where** ϕ states that when a concrete program is found that matches the parameterized program P_1, it should be transformed to P_2 if the side condition ϕ holds.

Parameterized programs. A parameterized program is a partially specified program that can represent multiple concrete programs. For example, in the original and transformed programs from Figure 2, $\mathbf{S_0}$ ranges over concrete statements (including branches, loops, and sequences of statements) that are single-entry-single exit; $\mathbf{I}$ ranges over concrete program variables; and $\mathbf{E}$ ranges over concrete expressions. Because variables like $\mathbf{S_0}$, $\mathbf{I}$ and $\mathbf{E}$ range over the syntax of concrete programs, we call such variables *meta-variables*. To simplify exposition, rather than provide explicit types for all meta-variables, we instead use the following naming conventions: meta-variables starting with $\mathbf{S}$ range over statements, meta-variables starting with $\mathbf{E}$ range over expressions, and meta-variables starting with $\mathbf{I}$ range over variables.

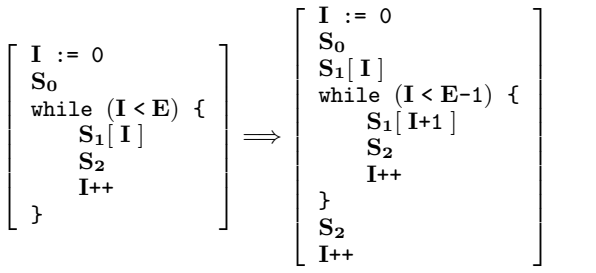


Figure 5. Software pipelining as one rewrite rule

Side Conditions. The side conditions are boolean combinations of facts that must hold at certain points in the original program. For example the side condition *DoesNotModify*$(\mathbf{S_0}, \mathbf{I})@L_1$ in Figure 2 states that at location L_1 in the original program $\mathbf{S_0}$ should not modify $\mathbf{I}$. In general, side conditions are first-order logic formulas with facts like *DoesNotModify*$(\mathbf{S_0}, \mathbf{I})@L_1$ as atomic predicates.

Each fact used in the side condition must have a semantic meaning, which is a predicate over program states. Figure 4 gives the semantic meanings for the three primary facts that we use in our system. In general, meanings can be first-order logic formulas with a few special function symbols: (1) σ is a term that represents the program state at the point where the fact holds. (2) $eval$ evaluates an expression in a program state and returns its value; (3) $step$ executes a statement in a program state and returns the resulting program state.

The semantic meanings are used by the PEC algorithm to determine the semantic information that can be inferred from the side conditions when proving correctness. Although optimization writers must provide these meanings, in our experience we have found that there is a small number of common facts used across many different optimizations (for example *DoesNotModify*), and since these meanings only need to be written once, the effort in writing meanings is not onerous.

Executing optimizations. Optimizations written in our language are meant to be executed by an execution engine. When running an optimization $P_1 \Longrightarrow P_2$ **where** ϕ, the execution engine must find concrete program fragments that match P_1. Furthermore, it must perform some program analysis to determine if the facts in the side condition ϕ hold. One option for implementing these program analyses is to use a general purpose programming language. Although this provides the most flexibility, it does not guarantee that the facts in the side condition are computed correctly. Alternatively, if one wants stronger correctness guarantees, the facts in the side conditions can be computed in a way that guarantees that their semantic meanings hold, for example using the Rhodium system of Lerner *et al.* [15], or using Leroy's Compcert system [16]. Although all the side conditions we used in our system pertain to the original program, it is also possible to express side conditions over the transformed program. In such cases, the execution engine would check the side conditions by building the transformed program before knowing that all the side conditions hold, and then

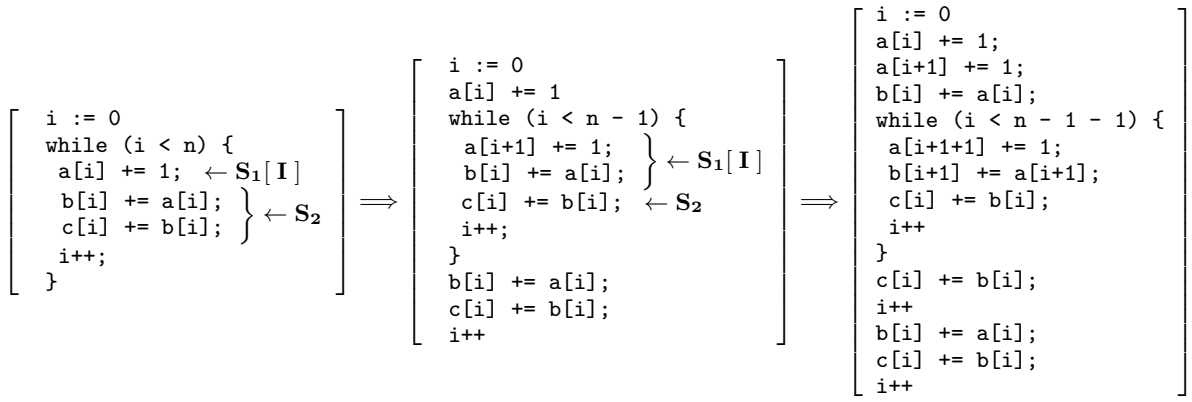


Figure 6. Two successive applications of the rewrite rule from Figure 5

running analyses on the transformed program to determine if the transformation can safely be performed.

The second part of software pipelining, shown in Figure 3, simply reorders statements. This optimization uses a new kind of meta-variable, $\mathbf{S_1}[\ \mathbf{I}\ ]$. In this case $\mathbf{S_1}$ is a program fragment with a hole in it, and $\mathbf{I}$ fills in the hole. The $\mathbf{S_1}[\ \mathbf{I}\ ]$ pattern is interpreted as follows: $\mathbf{S_1}$ matches any single-entry-single-exit piece of code that contains direct uses of a variable $\mathbf{I}$, but no modifications of $\mathbf{I}$. In such cases $\mathbf{S_1}$ gets matched to the code, but with holes wherever $\mathbf{I}$ occurs, so that $\mathbf{S_1}[\ \mathbf{I}{+}1\ ]$ represents the original statement with $\mathbf{I}$ replaced by $\mathbf{I}{+}1$. The fact that $\mathbf{I}$ is not modified in $\mathbf{S_1}$ and that $\mathbf{S_1}$ captures all uses of $\mathbf{I}$ allows us to treat such statements as function calls from a verification point of view.

The side condition in Figure 3 uses a new fact called *Commute*, which holds when two statements can be re-ordered. There are a variety of ways of implementing such a predicate when the compiler runs, for example the Omega test [21], or more generally dependence analysis [18]. We also show in Section 6 how we can express a version of *Commute* that can be written more easily in Rhodium, and thus proved correct automatically.

To ease presentation of how our software pipelining optimization operates, Figure 5 uses a single rewrite rule to summarize the effect of running the transformation from Figure 2 followed by the one from Figure 3. We show in Figure 6 how two applications of this single rewrite rule performs software pipelining on our example. At each step, we show what $\mathbf{S_1}$ and $\mathbf{S_2}$ are instantiated with. If there were more statements in the loop, the transformation from Figure 5 could be applied more times, with $\mathbf{S_1}$ ranging over one additional statement each time.

2.2 Proving Correctness of Software Pipelining

Our goal is to show that the software pipelining optimization written in our language is correct, once and for all, before it is even run once. To do this, we must show that each of the rewrite rules from Figures 2 and 3 satisfy the following property: given the side conditions, the original parameterized program and the transformed parameterized program have the same behavior. To illustrate the salient features of our approach, we show how we can prove the first and more complicated part of software pipelining, namely the rewrite rule from Figure 2.

Parameterized Equivalence Checking. Translation validation (TV) is a technique that has been used to prove equivalence of program fragments. Traditionally, TV is applied while the compiler is running, and so TV proves concrete, fully specified programs equivalent. In our setting, we are attempting to prove parameterized programs equivalent. To achieve this, we developed a technique called Parameterized Equivalence Checking (PEC) that generalizes traditional TV techniques to the setting of parameterized programs.

There are two simple observations that intuitively explain why techniques from translation validation can be generalized to parameterized programs. The first observation is that if a program fragment $\mathbf{S}$ in the original program executes in a program state σ, and the same program fragment $\mathbf{S}$ executes in the transformed program in the same state σ, then we know that the two resulting states are equal. This shows that we can reason about state equality even if we don't know what the program fragments are. The second observation is that when proving equivalence, we are usually interested in some key invariants that justify the optimization. The insight is that the semantic meaning of the side condition captures precisely when these key invariants can be propagated throughout statements that are not fully specified. For example, if the correctness of an optimization really depends on $\mathbf{I}$ not being modified in a region of code, the side condition will allow us to know this fact, and thus reason about $\mathbf{I}$ across such unknown statements.

Bisimulation relations. PEC proves equivalence using *bisimulation relations*, which are defined in terms of the more basic concept of *correlation relations*. A correlation relation is a set of entries, where each entry relates a program point in the original program with a corresponding program point in the transformed program. Each correlation relation entry also has a predicate that indicates how the state in the original program is related to the state of the transformed program at that point. A bisimulation relation is simply a correlation relation that satisfies the property that the predicate on any entry in the relation implies the predicate on all entries reachable from it.

The PEC approach works in two steps. In the first step we generate a correlation relation. In the second step, we check if the generated correlation relation has all the properties required to be a bisimulation relation, and if not, we iteratively strengthen it until it does.

Figure 7 shows the control flow graph (CFG) of the original and the transformed programs in our example, along with the correlation relation that our approach generates. The entries in the relation are labeled A through G, and each entry has a predicate associated with it. These predicates operate over the program states σ_1 and σ_2 of the original and transformed program. To make the notation cleaner, we use some shorthand notation. For example, $\mathbf{E}_1$ means $eval(\sigma_1, \mathbf{E})$. Using this notation, the predicate at edge D states that (1) the two programs states σ_1 and σ_2 are equal, (2) $\mathbf{I} < \mathbf{E}$ holds

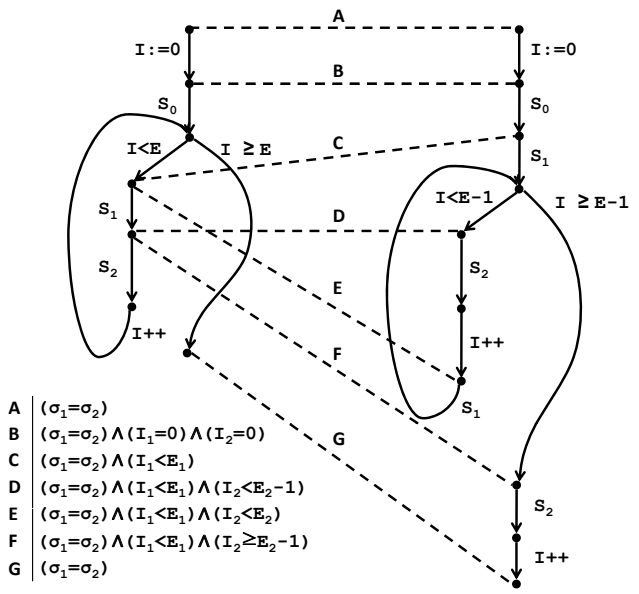


Figure 7. CFGs of running example with the correlation relation

in σ_1 and (3) $\mathbf{I} < \mathbf{E} - 1$ holds in σ_2. In this example, our approach would determine that the generated correlation relation is in fact a bisimulation relation and as a result does not need to be strengthened.

Generating a correlation relation. To generate the correlation relation, our PEC algorithm first adds edge A with predicate $\sigma_1 = \sigma_2$; this indicates that we can assume the program states are equal at the top of the code. It also adds edge G with predicate $\sigma_1 = \sigma_2$; this indicates that we must establish that the program states are equal after the two programs execute. To generate the points in between, our algorithm traverses both programs in parallel from the top entry. Each time a statement is reached like $\mathbf{S_0}$, $\mathbf{S_1}$, and $\mathbf{S_2}$, the algorithm finds the corresponding point in the other program, and adds a relation entry between the two statements with the predicate $\sigma_1 = \sigma_2$ (since this is the only mechanism we have to preserve equivalence of arbitrary statements). Finally, our algorithm for generating a correlation relation strengthens the predicate along each side of a branch with the branch condition, which leads to the various conditions relating $\mathbf{I}$ and $\mathbf{E}$ in Figure 7. This allows entries in the bisimulation relation to encode information about what branch conditions they are under.

Strengthening the correlation relation. Once a correlation relation is generated, our PEC algorithm checks whether or not it is a bisimulation relation, and if not, iteratively strengthens it. In particular, we must show that the predicate at a given entry is strong enough to imply the predicates at all entries reachable from it. To achieve this, our approach traverses the two programs in parallel starting at each correlation relation entry to find the next reachable entries. Then, for each discovered path between a correlation relation entry X and another entry Y, our algorithm asks a theorem prover to show that, if the two programs start executing at X in states where X's predicate holds, and Y is reached, then Y's predicate holds.

In our example from Figure 7, the paths that our algorithm would discover between correlation relation entries are as follows: A to B, B to C, C to F, C to D, D to E, E to D, E to F, and F to G.

As an example, for the B-C path, our tool would ask a theorem prover to show that, for any σ_1 and σ_2: if (1) $\sigma_1 = \sigma_2$ holds and (2) the original program executes $[\mathbf{S_0}; \texttt{assume}(\mathbf{I} < \mathbf{E})]$ and (3) the transformed program executes $[\mathbf{S_0}]$, then $\sigma_1 = \sigma_2 \wedge eval(\sigma_1, \mathbf{I}) < eval(\sigma_1, \mathbf{E})$ will hold after the two statements have executed. In this case, the implication follows immediately from the `assume` and the fact that $\mathbf{S_0}$ produces the same program state if started in the same program state. If for some path pair X to Y, the implication does not hold (this is not the case in Figure 7), our algorithm would strengthen the condition at X with the weakest precondition of the condition at Y. Using such iterative strengthening, our algorithm tries to convert the original correlation relation into a bisimulation relation.

Our algorithm must also prune infeasible paths when it performs the checks. For example, when starting at F, it is impossible for the original program to stay in the loop – it must exit to G. Our checker can determine this from the predicate at F. In particular, let i and e be the original values of $\mathbf{I}$ and $\mathbf{E}$ at F (in either σ_1 or σ_2 since they are equal). The value e does not change through the loop as stated by the side conditions. If the original program chooses to stay in the loop, the $\texttt{assume}(\mathbf{I} < \mathbf{E})$ would give us $i + 1 < e$ (where the "+1" comes from the increment of $\mathbf{I}$ and the fact that $\mathbf{S_2}$ does not modify $\mathbf{I}$). This would be inconsistent with the assumption from F stating that $i \geq e - 1$, and thus the path is pruned.

2.3 Benefits or our approach

Our PEC approach has several benefits over previous techniques for proving optimizations correct in a fully automated way. First, PEC can prove the correctness of complex rewrite rules once and for all, something that previous systems like Rhodium were not able to do. This allows us to implement and prove many more optimizations correct, as discussed in Section 7, thus improving the state of the art of the set of optimizations that can be proved automatically once and for all.

Second, because parameterized programs can contain concrete statements, PEC can in fact prove fully concrete programs equivalent (with no side conditions provided). In this setting, our PEC technique subsumes many previous approaches to translation validation, for example the relation approach of Necula [19] and the permute approach of Zuck *et al.* [6].

Finally, PEC enables a new staged paradigm for optimization verification: we can use PEC to check as many of the optimizations as possible before they are run. For those optimizations that we can't prove correct once and for all, we can use PEC again when the compiler is running to perform translation validation on the concrete input/output pairs.

3. Parameterized Equivalence Checking

We now describe our approach in more detail. Our goal is to show that two parameterized programs P_1 and P_2 are equivalent under side conditions ϕ. We represent each program P as a Control Flow Graph (CFG), which we denote by π. In particular, we assume that π_1 is the CFG of the original program, and π_2 is the CFG of the transformed program. Each node in a CFG is a program location l, and edges between program locations are labeled with statements. We use ι_1 and ι_2 to denote the entry locations of π_1, π_2, and ϵ_1, ϵ_2 to represent the exit locations of π_1, π_2.

Given a program state σ, we use the notation $\pi(\sigma)$ to represent the program state after executing π starting in state σ.

DEFINITION 1 (Equivalence). *Given two programs π_1 and π_2, we define π_1 to be equivalent to π_2 if for any program state σ, we have $\pi_1(\sigma) = \pi_2(\sigma)$.*

The above definition of equivalence allows us to use our optimizations anywhere inside in a program: by establishing program state equivalence, we guarantee that the remainder of the program, after

the optimization, runs the same way in the original and the transformed programs. We can model observable events such as IO using heap updates. For example, a call to `printf` can just append its arguments to a linked list on the heap. In this setting, our approach guarantees that the order of IO events is preserved.

The statements in our CFGs are taken from a concrete programming language of statements, extended with meta-variables. The main components of our approach do not depend on the choice of the concrete language of statements that we start with: this language can for example include pointers, arrays, and function calls. We do however make one exception to this rule: we assume the existence of `assume` statements. In particular, we model conditionals using `assume` statements on the edges that flow away from a branch location (as shown for example in Figure 7). We also use `assume` statements to insert the information from side conditions into the original or transformed program as needed, so that our tool can reason about the side conditions. The choice of concrete language only affects the semantics of statements, which is entirely modularized in a function called $step$ (which we have already seen). The only part of the system that knows about $step$ is the theorem prover, which is given background axioms about the semantics of instructions (so that it knows for example how **I++** updates the store). All other parts of the system treat $step$ as a black box.

Bisimulation relations. Our approach is based on using a bisimulation relation to relate the execution of the original program and the transformed program. Before defining what a bisimulation relation is, we first define a more basic concept, which is a correlation relation $\mathcal{R}$. A correlation relation $\mathcal{R}$ is a set of triples of the form (l_1, l_2, ψ), where l_1 is a location in π_1, l_2 is a location in π_2, and ψ is a formula relating the program states at l_1 and l_2. In particular, ψ is a formula over σ_1 and σ_2, the states in the original and transformed program respectively.

Since ψ is a predicate with free variables σ_1 and σ_2, we can use ψ as a function from two program states to booleans. We use the notation $l \in \mathcal{R}$ to mean $(l, \_, \_) \in \mathcal{R}$ or $(\_, l, \_) \in \mathcal{R}$, where $\_$ is a wildcard that pattern matches anything. If $l \in \mathcal{R}$, we say that l *occurs in* $\mathcal{R}$. We define a relation $\longrightarrow_{\mathcal{R}}$ which is the successor relation on CFGs, except that it skips locations not in $\mathcal{R}$. In order to define just one relation $\longrightarrow_{\mathcal{R}}$ for both CFGs π_1 and π_2, we assume without loss of generality that the two CFGs π_1 and π_2 have disjoint locations. With this assumption, $\longrightarrow_{\mathcal{R}}$ is defined as: $l \xrightarrow{p}_{\mathcal{R}} l'$ holds iff $l \in \mathcal{R}$ and $l' \in \mathcal{R}$ and there is path p in the CFG of π_1 or π_2 from l to l' such that none of the locations on the path, except end points, occur in $\mathcal{R}$.

Simulation and bisimulation relations are correlation relations with some additional properties. Below we define these relations by adapting previous work [17] to the setting of CFGs.

DEFINITION 2 (Simulation Relation). *A correlation relation $\mathcal{R}$ is a simulation relation for π_1, π_2 iff it satisfies the following properties:*

1. $(\iota_1, \iota_2, \sigma_1 = \sigma_2) \in \mathcal{R}$ *and* $(\epsilon_1, \epsilon_2, \sigma_1 = \sigma_2) \in \mathcal{R}$
2. *for any* $l_1, l_1', l_2, p_1, \psi$, *if* $(l_1, l_2, \psi) \in \mathcal{R}$ *and* $l_1 \xrightarrow{p_1}_{\mathcal{R}} l_1'$ *then there exists* l_2', ψ', p_2 *such that* $(l_1', l_2', \psi') \in \mathcal{R}$ *and* $l_2 \xrightarrow{p_2}_{\mathcal{R}} l_2'$ *and* $\forall \sigma_1, \sigma_2 \,.\, \psi(\sigma_1, \sigma_2) \Rightarrow \psi'(step(\sigma_1, p_1), step(\sigma_2, p_2))$.

DEFINITION 3 (Bisimulation Relation). *A correlation relation $\mathcal{R}$ is a bisimulation relation for π_1, π_2 iff $\mathcal{R}$ is a simulation relation for π_1, π_2 and $\mathcal{R}^{-1}$ is a simulation relation for π_2, π_1, where $\mathcal{R}^{-1}$ is defined by $(l_1, l_2, \psi) \in \mathcal{R}$ iff $(l_2, l_1, \psi) \in \mathcal{R}^{-1}$.*

THEOREM 1 (Bisimulation Equivalence). *If there exists a bisimulation relation between π_1 and π_2 then π_1 and π_2 are equivalent.*

function $\mathsf{PEC}(\pi_1, \pi_2, \phi)$
  let $(\pi_1', \pi_2') := \mathsf{Permute}(\pi_1, \pi_2, \phi)$
  let $\mathcal{R} := \mathsf{Correlate}(\pi_1', \pi_2')$
  return $\mathsf{Check}(\mathcal{R}, \pi_1', \pi_2', \phi)$

Figure 8. Parameterized Equivalence Checking

The conditions from Definition 2 are the base case and the inductive cases of a proof by induction showing that π_1 is equivalent to π_2. Thus, a bisimulation relation is a witness that two CFGs are equivalent. Our approach is based on the above theorem. In particular, our general approach is to try to infer a bisimulation relation to show that π_1 and π_2 are equivalent.

Architectural overview. Figure 8 shows the pseudo-code of our PEC approach. There are three steps: the Permute module, the Correlate module and the Checker module. The Permute module runs as a pre-processor before we use our main bisimulation-based approach. The Permute module applies a general form of the Permute theorem that has been used in translation validation of loop optimizations [6], but it does so on parameterized programs. After the Permute module has run, the Correlate and Checker module implement our bisimulation approach. In particular, the Correlate module first generates a correlation relation $\mathcal{R}$ from the two CFGs π_1 and π_2. The Checker module then makes sure that the properties from Definitions 2 and 3 hold, possibly strengthening the relation in order to guarantee property 2. The next three sections of the paper describe each of the modules in our system. We first describe the Correlate and Checker modules, which are at the heart of our approach, and then move on the Permute module, which acts as a preprocessing step.

4. Correlate module

To prove that two parameterized programs are equivalent our approach attempts to discover a bisimulation relation between them. To do this, the Correlate module computes a correlation relation, which will then be strengthened to a bisimulation relation by the Checker module.

Two kinds of locations in π_1 and π_2 are particularly important while constructing the correlation relation $\mathcal{R}$: locations that immediately precede a statement meta-variable, the set of which we denote L_S, and locations that immediately precede an `assume`, the set of which we denote L_A. We define the $\rightarrow_S$ and $\rightarrow_A$ relations to be the successor relation in the CFG, but skipping over nodes that are not in L_S or L_A, respectively. We assume that the two CFGs π_1 and π_2 have disjoint locations, which means we can use a single version of L_S that applies to both CFGs (and similarly for L_A). More precisely: (1) $l \rightarrow_S l'$ holds iff $l' \in L_S$ and there exists a path from l to l' in π_1 or π_2 with no intermediate locations in L_S, and (2) $l \xrightarrow{p}_A l'$ holds iff p is a path from l to l' in π_1 or π_2 that has no intermediate locations in L_A and $l \in L_A \cup \{\iota_1, \iota_2\}$.

Using these definitions, the correlation relation that we compute is the smallest relation $\mathcal{R}$ such that:

$$\mathcal{R}(\iota_1, \iota_2, \sigma_1 = \sigma_2) \wedge \mathcal{R}(\epsilon_1, \epsilon_2, \sigma_1 = \sigma_2) \tag{1}$$

$$\forall l_1, l_2, l_1', l_2' \,. \left(\begin{array}{l} \mathcal{R}(l_1, l_2, \_) \wedge \\ l_1 \rightarrow_S l_1' \wedge l_2 \rightarrow_S l_2' \wedge \\ (l_1', l_2') \neq (\epsilon_1, \epsilon_2) \end{array} \right) \Rightarrow \mathcal{R}(l_1', l_2', \mathit{Cond}(l_1', l_2')) \tag{2}$$

where $\mathit{Cond}(l_1, l_2) = \mathit{Post}(l_1) \wedge \mathit{Post}(l_2) \wedge \sigma_1 = \sigma_2$

and $\mathit{Post}(l) = \bigvee_{\{l' \,|\, l \xrightarrow{p}_A l'\}} \mathsf{SP}(p, true)$

```
1.  function Check(R, π1, π2, φ)
2.      let R := R ∪ {(ι1, ι2, σ1 = σ2), (ε1, ε2, σ1 = σ2)}
3.      let (π1', π2') := InsertAssumes(π1, π2, φ)
4.      let P := ComputePaths(R, π1', π2')
5.      if P = Fail then return Fail
6.      let C := GenerateConstraints(P)
7.      return SolveConstraints(C, R)

8.  function ComputePaths(R, π1, π2)
9.      let P := ∅
10.     for each (l1, l2, ψ) ∈ R
11.         for each p1, l1' such that l1 --p1-->R l1' do
12.             for each p2, l2' such that l2 --p2-->R l2' do
13.                 if ¬Infeasible(p1, p2, ψ)
14.                     if (l1', l2', _) ∉ R then return Fail
15.                     P := P ∪ {(l1, l2, p1, p2, l1', l2')}
16.     return P

17. function Infeasible(p1, p2, ψ)
18.     return ATP(¬(SP(p1, ψ) ∧ SP(p2, ψ))) = Valid

19. function GenerateConstraints(P)
20.     let C := ∅
21.     for each (l1, l2, p1, p2, l1', l2') ∈ P do
22.         C := C ∪ {X(l1,l2) ⇒ PWP(p1||p2, X(l1',l2'))}
23.     return C

24. function SolveConstraints(C, R)
25.     let soln := map from constraint vars to formulas
26.     for each (l, l', ψ) ∈ R do soln(X(l,l')) := ψ
27.     let worklist := C
28.     while worklist not empty do
29.         let [Xx ⇒ PWP(p1||p2, Xy)] := worklist.remove
30.         let F := [soln(Xx) ⇒ PWP(p1||p2, soln(Xy))]
31.         if ATP(F) ≠ Valid then
32.             if x = (ι1, ι2) then return Fail
33.             soln(Xx) := soln(Xx) ∧ PWP(p1||p2, soln(Xy))
34.             worklist := worklist ∪
35.                 {c ∈ C | c = [ _ ⇒ PWP( _ , Xx)]}
36.     return Success
```

Figure 9. Pseudo-code for the Checker module

Here *Cond*(l_1, l_2) computes the formula over σ_1 and σ_2 that should hold when π_1 and π_2 are at locations l_1 and l_2 respectively. Within *Cond*, the predicate *Post*(l) is the disjunction of the strongest post conditions with respect to $true$ over paths p for which there exists some l' such that $l' \xrightarrow{p}_A l$.

The Correlate function from Figure 8, which is the core of the Correlate module, computes the correlation relation using the above definition. In particular, it starts with an empty relation, and first applies Formula (1) to correlate the entry and exit nodes. Then it iteratively applies Formula (2) until no more entries can be added.

5. Checker module

The pseudo-code for the Checker module is shown in Figure 9. The checker performs the following five steps: first, it makes sure that the entry and exit locations are related with full state equality (line 2); then it inserts assume statements into the original and transformed programs corresponding to the side conditions that are given in the rewrite rule (line 3); then it computes the paths between entries in the correlation relation, doing path pruning anywhere possible (line 4); using the computed paths it generates a set of constraints that the final correlation relation must satisfy to be a bisimulation relation (line 6); finally it solves the generated constraints using a fixed point computation (line 7). We describe each of these steps in more detail.

InsertAssumes. The InsertAssumes function inserts the side condition assumptions into the original and transformed programs in the form of `assume` statements. An `assume` statement takes as argument a predicate over the program state σ that occurs at the point where the assume holds. To ease presentation, we make the simplifying assumption that $\phi = \phi_1@L_1 \wedge ... \wedge \phi_n@L_n$ (our implementation handles the general case). For each side condition ϕ_i, we define $[\![\phi_i]\!]$ to be a predicate over σ that directly encodes the side condition's meaning provided by the optimization writer. Then for each $\phi_i@L_i$, we find the location L_i in either the original or the transformed program, and insert `assume`$([\![\phi_i]\!])$ at that location.

ComputePaths. The ComputePaths function computes the set of paths $\mathcal{P}$ between entries of the correlation relation $\mathcal{R}$. The function starts by initializing the set of paths to the empty set (line 9). Then, for each correlation relation entry $(l_1, l_2, \psi) \in \mathcal{R}$, it finds the reachable program points l_1' and l_2' in each program that are in the correlation relation (lines 10-12). It does so using the $l \xrightarrow{p}_{\mathcal{R}} l'$ relation introduced in Section 3, which states that l occurs in $\mathcal{R}$, l' occurs in $\mathcal{R}$, and there is a CFG path p from l to l' where none of the locations in the path, except for the end points, occur in $\mathcal{R}$. For each pair of paths p_1, p_2 that are found, ComputePaths checks if the paths are infeasible by calling the Infeasible function. Infeasible first computes the strongest postcondition of p_1 and p_2. If an automated theorem prover (ATP) can show that the two post-conditions are inconsistent, then the combination of those two paths is infeasible, and can be pruned. The Infeasible function performs the pruning that was intuitively described for the software pipelining example in Section 2.2. If the paths are feasible and an entry $(l_1', l_2', \_)$ exists in the correlation relation, then the two paths are added to $\mathcal{P}$, along with the beginning and end points (line 15).

GenerateConstraints. Once the set of paths in the correlation relation have been collected, the GenerateConstraints function computes the set of constraints $\mathcal{C}$ that our correlation relation must satisfy to be a bisimulation relation. For each $(l, l', \_) \in \mathcal{R}$, we define a constraint variable $\mathcal{X}_{(l,l')}$ that represents the formula in the correlation relation relating l and l'. Then, for each path between two entries in the correlation relation (line 21), we add a constraint stating that the predicate at the beginning of the path must imply the predicate at the end of the path (line 22). We express this condition using the weakest precondition computation PWP, which is a parallel version of the regular weakest precondition.

The main challenge in expressing this weakest precondition is that the traditional formulation of weakest precondition depends on the structure of the statements being processed. As a result, it is difficult to use this definition for statements like $\mathbf{S_0}$ and $\mathbf{S_1}$ in our parameterized programs, because the precise structure of these statements is not known. To address this challenge, we use an alternate yet equivalent definition of weakest precondition. In particular, consider the traditional weakest precondition computation, and assume that the predicate we are computing is a function from program states to booleans. Then the traditional weakest precondition WP can be expressed as:

$$\mathsf{WP}(\mathbf{S}, \psi)(\sigma) = \psi(step(\sigma, \mathbf{S}))$$

If we assume that the program state σ is simply a free variable in the predicate ψ, then WP can be expressed as:

$$\mathsf{WP}(\mathbf{S}, \psi) = \psi[\sigma \mapsto step(\sigma, \mathbf{S})]$$

Generalizing this to two parallels paths in two different programs, the predicates now have free variables σ_1 and σ_2, and we can express PWP as follows:

$$\mathsf{PWP}(p_1||p_2, \psi) = \psi[\sigma_1 \mapsto step(\sigma_1, p_1), \sigma_2 \mapsto step(\sigma_2, p_2)]$$

SolveConstraints. Once the set of constraints have been generated, the SolveConstraints function tries to solve these constraints iteratively by starting with the correlation relation $\mathcal{R}$, and iteratively strengthening the conditions in the relation until all the constraints are satisfied. In particular, SolveConstraints maintains a map *soln* that maps each constraint variable to the formula we currently associate the variable with. The *soln* map is initialized with the predicates from the correlation relation (line 26). SolveConstraints also maintains a worklist of constraints to be processed, which is initialized with all the constraints (line 27). While the worklist is not empty, SolveConstraints removes a constraint from the worklist (line 29), and if the constraint is not satisfied (line 31), it strengthens the left-hand side of the implication in the currently stored solution (line 33), and adds to the worklist all the constraints that need to be checked again because of the strengthening (line 35). One subtlety is that we cannot strengthen the relation at the entry points ι_1, ι_2. If we ever try to do this, we indicate a failure (line 32). Because SolveConstraints is trying to compute a fixedpoint over the very flexible but infinite domain of boolean formulas, it may not terminate. However, as our experiments show in Section 7, in practice SolveConstraints quickly finds a fixed point.

6. Permute module

Our main technique for PEC relies on a bisimulation approach to prove equivalence. However, the bisimulation approach has some known limitations. In particular, bisimulation relations are not well suited for proving the correctness of non-structure preserving transformations, which are transformations that change the execution order of code across loop iterations. Previous work on translation validation has devised a technique called Permute [33] for handling such transformations on concrete programs. We have adapted this technique to the setting of parameterized programs.

Our version of Permute runs as a pre-pass to our bisimulation relation approach: Permute looks for loops in the original and transformed programs that it can prove equivalent, and for the ones it can, it replaces them with a new fresh variable $\mathbf{S}$, which will then allow the bisimulation relation part of our PEC approach to see that they are equivalent.

Our Permute algorithm tries to find a general nested loop of the following form, where we use $\prec_L$ to denote a total order on L

$$\begin{array}{c}\texttt{for } i_1 \in I_1 \texttt{ by } \prec_{I_1} \texttt{ do} \\ \ddots \\ \texttt{for } i_n \in I_n \texttt{ by } \prec_{I_n} \texttt{ do} \\ B(i_1, \cdots, i_n)\,;\end{array}$$

where I_j is the domain of the index variable i_j

The relation $\prec_{I_j}$ represents the order in which the index variable i_j is traversed. The above general nested loop can be represented more compactly as follows:

$$\texttt{for } \vec{i} \in \vec{I} \texttt{ by } \prec_{\vec{I}} \texttt{ do } B(\vec{i})\,;$$

where $\vec{I} = I_1 \times \cdots \times I_n$ **and**

$$\vec{i} \prec_{\vec{I}} \vec{j} \iff \bigvee_{k=1}^{n} (i_1, \cdots, i_{k-1}) = (j_1, \cdots, j_{k-1}) \wedge i_k \prec_{I_k} j_k$$

The relation $\prec_{\vec{I}}$ above is the lexicographic order on $\vec{I}$.

Our algorithm tries to find a loop structure as above in the original program and in the transformed program, and for each such pair, it tries to show that the following loop reordering transformation is correct:

$$\begin{array}{c}\texttt{for } \vec{i_1} \in \vec{I_1} \texttt{ by } \prec_{\vec{I_1}} \texttt{ do } B(\vec{i_1}) \\ \Downarrow \\ \texttt{for } \vec{i_2} \in \vec{I_2} \texttt{ by } \prec_{\vec{I_2}} \texttt{ do } B(F(\vec{i_2}))\end{array} \tag{3}$$

The above transformation may change the order of the index variables by changing the domain $\vec{I_1}$ to $\vec{I_2}$ and the relation $\prec_{\vec{I_1}}$ to $\prec_{\vec{I_2}}$ and also possibly changing the loop's body by applying a linear transformation from $B(\vec{i_1})$ to $B(F(\vec{i_2}))$.

To show that the above transformation is correct, we need to ensure that the transformed loop executes the same instances of the loop body in an order that preserves the body's behavior. In order to define the conditions under which this happens, we first define when two program fragments commute.

DEFINITION 4 (Commute). *We say two program fragments S_1 and S_2 commute, written $S_1 \approx S_2$, if starting from an arbitrary initial state, the resultant state of executing S_1 and then S_2 is the same as executing S_2 and then S_1.*

We can now guarantee that the original and transformed loops are equivalent by requiring the following properties to hold:

1. There exists a 1-1 correspondence between $\vec{I_1}$ and $\vec{I_2}$.
2. For every $\vec{i_1}, \vec{i_2} \in \vec{I_1}$, if $B(\vec{i_1})$ executes before $B(\vec{i_2})$ in the original program and $B(\vec{i_2})$ executes before $B(\vec{i_1})$ in the transformed program then $B(\vec{i_1})$ and $B(\vec{i_2})$ commute, i.e. $B(\vec{i_1}) \approx B(\vec{i_2})$

The first property above can be established by showing that the linear function $F : \vec{I_2} \longrightarrow \vec{I_1}$ is a bijective function, i.e. F is one-to-one and onto. This in turn can be guaranteed by defining an inverse function $F^{-1} : \vec{I_1} \longrightarrow \vec{I_2}$. The above observations are summarized in the following Permute Theorem.

THEOREM 2 (Permute). *A loop reording transformation of the form shown in Formula* (3) *preserves semantics if the following hold:*

1. $\forall \vec{i_2} \in \vec{I_2}.$ $\quad F(\vec{i_2}) \in \vec{I_1}$
2. $\forall \vec{i_1} \in \vec{I_1}.$ $\quad F^{-1}(\vec{i_1}) \in \vec{I_2}$
3. $\forall \vec{i_2} \in \vec{I_2}.$ $\quad \vec{i_2} = F^{-1}(F(\vec{i_2}))$
4. $\forall \vec{i_1} \in \vec{I_1}.$ $\quad \vec{i_1} = F(F^{-1}(\vec{i_1}))$
5. $\forall \vec{i_1}, \vec{i_1'} \in \vec{I_1}.$ $\quad \vec{i_1} \prec_{\vec{I_1}} \vec{i_1'} \wedge F^{-1}(\vec{i_1'}) \prec_{\vec{I_2}} F^{-1}(\vec{i_1}) \implies B(\vec{i_1}) \approx B(\vec{i_1'})$

Theorem 2 was introduced and proved in previous work [33, 21, 23]. The Permute module tries to apply Theorem 2 by asking an automated theorem prover to discharge the preconditions of the theorem assuming the side conditions given in the transformation. As an example, consider the simple loop interchange optimization shown in Figure 10. For clarity and ease of explanation, the example is simplified here to have constant bounds (L_1, U_1, L_2, U_2) instead of arbitrary expressions. However, our tool checks the more general version of this example.

$$\left[\begin{array}{l}\texttt{for } (\mathbf{I} \texttt{ := } \mathbf{L_1};\ \mathbf{I} \le \mathbf{U_1};\ \mathbf{I}\texttt{++}\ )\ \{ \\ \quad \texttt{for } (\mathbf{J} \texttt{ := } \mathbf{L_2};\ \mathbf{J} \le \mathbf{U_2};\ \mathbf{J}\texttt{++})\ \{ \\ L_1 : \quad\ \mathbf{S}[\ \mathbf{I}, \mathbf{J}\ ] \\ \quad \} \\ \}\end{array}\right]$$

$$\Downarrow$$

$$\left[\begin{array}{l}\texttt{for } (\mathbf{J} \texttt{ := } \mathbf{L_2};\ \mathbf{J} \le \mathbf{U_2};\ \mathbf{J}\texttt{++})\ \{ \\ \quad \texttt{for } (\mathbf{I} \texttt{ := } \mathbf{L_1};\ \mathbf{I} \le \mathbf{U_1};\ \mathbf{I}\texttt{++})\ \{ \\ \qquad \mathbf{S}[\ \mathbf{I}, \mathbf{J}\ ] \\ \quad \} \\ \}\end{array}\right]$$

where $\forall\, \mathbf{K}, \mathbf{L}\ .\ (\mathbf{K} \neq \mathbf{I} \wedge \mathbf{L} \neq \mathbf{J}) \Rightarrow$

$$\left(\begin{array}{l}\textit{DoesNotModify}(\mathbf{S}[\ \mathbf{I}, \mathbf{J}\ ], \mathbf{S}[\ \mathbf{K}, \mathbf{L}\ ])@L_1 \wedge \\ \textit{DoesNotModify}(\mathbf{S}[\ \mathbf{K}, \mathbf{L}\ ], \mathbf{S}[\ \mathbf{I}, \mathbf{J}\ ],)@L_1 \wedge \\ \textit{DoesNotInterfere}(\mathbf{S}[\ \mathbf{I}, \mathbf{J}\ ], \mathbf{S}[\ \mathbf{K}, \mathbf{L}\ ])@L_1\end{array}\right)$$

fact *DoesNotModify*$(\mathbf{S_1}, \mathbf{S_2})$
has meaning $step(\sigma, \mathbf{S_2})|^{\sigma}_{\mathbf{S_2}} = step(step(\sigma, \mathbf{S_1}), \mathbf{S_2})|^{\sigma}_{\mathbf{S_2}}$
fact *DoesNotInterfere*$(\mathbf{S_1}, \mathbf{S_2})$
has meaning $step(\sigma, \mathbf{S_2})|^{\sigma}_{\mathbf{S_2}} = step(step(\sigma, \mathbf{S_2}), \mathbf{S_1})|^{\sigma}_{\mathbf{S_2}}$

Figure 10. Loop Interchange

The Permute module first transforms the original and transformed programs into our canonical representations of loops. In particular, the original program is summarized as

$$\vec{I_1} = \{(i,j) \mid i \in [\mathbf{L_1}, \mathbf{U_1}], j \in [\mathbf{L_2}, \mathbf{U_2}]\}$$
$$\textbf{and}\quad B((i,j)) = \mathbf{S}[\ \mathbf{i}, \mathbf{j}\ ]$$
$$\textbf{and}\quad \prec_{\vec{I_1}} \text{ is the lexicographic order on } \vec{I_1}$$

and the transformed program is represented as

$$\vec{I_2} = \{(i,j) \mid i \in [\mathbf{L_2}, \mathbf{U_2}], j \in [\mathbf{L_1}, \mathbf{U_1}]\}$$
$$\textbf{and}\quad B((i,j)) = \mathbf{S}[\ \mathbf{j}, \mathbf{i}\ ]$$
$$\textbf{and}\quad \prec_{\vec{I_2}} \text{ is the lexicographic order on } \vec{I_2}$$

Since there is one loop in the original program and one in the transformed program, Permute tries to prove them equivalent. In order to apply the Permute Theorem, our tool needs to infer the two mapping functions F and F^{-1}, and prove properties 1 through 4 of Theorem 2. Permute infers these functions automatically using a simple heuristic that runs a range analysis over the original and transformed programs, and uses the results of the upper and lower bounds on index variables to infer F and F^{-1}. For our loop interchange optimization, our tool automatically infers that the two functions are: $F((i,j)) = (j,i)$, and $F^{-1}((i,j)) = (j,i)$. Our heuristic infers the appropriate mapping functions in all the optimizations that we have tried (see Section 7). However, we also provide the ability for the programmer to provide F and F^{-1} in the case where our heuristic cannot find appropriate functions.

The purpose of the side conditions of loop interchange is to allow the theorem prover to show property 5 of Theorem 2. One option for expressing the side condition is to use the *Commute* fact from Figure 4, which gives us a predicate very close to property 5 directly, and then use a heavyweight analysis when the compiler runs to establish *Commute* (for example a theorem prover, the Omega test [21], or more generally dependence analysis [18]). Another option, which we use in Figure 10 to illustrate the flexibility of our approach, is to use a more syntactic definition of commutativity, using two new facts: *DoesNotModify*, which holds when a statement does not modify the variables or heap locations that another may read, and *DoesNotInterfere*, which holds when a statement does not modify the variables or heap locations that another may write to. The notation $\sigma_1|^{\sigma_2}_{\mathbf{S}}$ represents the state σ_1 projected onto the variables and heap locations that $\mathbf{S}$ modifies if it executes starting in state σ_2. The benefit of using the more syntactic *DoesNotModify* and *DoesNotInterfere* facts is that they can more easily be implemented using simple Rhodium dataflow functions, which in turn can be proved correct automatically. In this way we will know that the computed facts when the compiler runs imply the semantic meanings that our PEC technique assumed when proving the correctness of loop interchange once and for all.

Optimizations	**Uses permute**	**Time (secs)**	**# ATP calls**
Category 1			
Copy propagation	No	1	3
Constant propagation	No	1	3
Common sub-expression elim	No	1	3
Partial redundancy elimination	No	3	13
Category 2			
Loop invariant code hoisting	No	8	25
Conditional speculation	No	2	14
Speculation	No	3	12
Category 3			
Software pipelining	No	5	19
Loop unswitching	No	16	94
Loop unrolling	No	10	45
Loop peeling	No	6	40
Loop splitting	No	15	64
Loop alignment	Yes	1	5
Loop interchange	Yes	1	5
Loop reversal	Yes	1	5
Loop skewing	Yes	2	5
Loop fusion	Yes	4	10
Loop distribution	Yes	4	10

Figure 11. Optimizations proven correct using PEC. Category 1: expressible and provable in Rhodium; Category 2: provable in Rhodium, but our version is more general and easier to express; Category 3: not expressible or provable in Rhodium.

7. Evaluation

We implemented PEC in 2,408 lines of OCaml using the Simplify theorem prover [5] to realize the ATP module from Figure 9.

Figure 11 lists a selection of optimizations that we proved correct using our implementation. For each optimization we list the time it took to carry out PEC and the number of queries to the theorem prover. To be clear about our contribution compared to the Rhodium system for automatically proving optimizations correct, Figure 11 partitions the optimizations into three categories.

Category 1 : Optimizations that were also expressed and proved correct in Rhodium, and whose PEC formulation is equivalent to the Rhodium formulation.

Category 2 : Optimizations that could have been expressed and proved correct in Rhodium, but our versions are much more general than the Rhodium version, and also much easier to express. For example, in the case of loop invariant code hoisting, PEC can prove the correctness of hoisting loop-invariant branches or even entire loops, while the Rhodium version could only hoist loop-invariant assignments. Furthermore, these optimizations are much easier to express in our PEC formulation because of our explicit support

for many-to-many rewrites. In contrast, implementing these optimizations in Rhodium would require an expert to carefully craft sequences of local statement rewrites that achieves the intended effect. For example, moving a statement in Rhodium requires inserting a duplicate copy of the statement at the target location, and then removing the original statement in a separate pass.

Category 3 : Optimizations that cannot be proved correct, or even expressed, in Rhodium. Our support for many-to-many rewrite rules makes it easy to express these optimizations, and PEC technique is general enough to handle their correctness proofs.

The trusted computing base for our system includes: (1) the PEC checker, comprising 2,408 lines of OCaml code (2) the Simplify automated theorem prover, a widely used and well tested theorem prover, and (3) the execution engine that will run the optimizations. Within the execution engine, the trust can be further subdivided into two components. The first component of the execution engine must perform the syntactic pattern matching for rewrite rules, and apply rewrite rules when they fire. This part is always trusted. The second component of the execution engine must perform program analyses to check each optimization's side-conditions in a way that guarantees their semantic meaning. Here our system offers a choice. These analyses can either be trusted and thus implemented inside the compiler using arbitrarily complex analyses, or untrusted and implemented using a provably safe analysis system like Rhodium.

8. Execution Engine

We implemented a prototype execution engine in 383 lines of OCaml code that runs optimizations checked by PEC. Although PEC can be applied to any intermediate representation for which we can compute weakest preconditions, our prototype execution engine transforms programs written in a C-like intermediate language including arrays and function calls. Using this prototype, we were able to run all the optimizations described in previous sections. Even though our execution engine is a prototype, it demonstrates how our optimizations can be incorporated into a compiler, and also shows that the optimizations we checked execute as expected.

Our execution engine is embodied in a function called $Apply$, which takes as input a program p, a transformation rule $[P_1 \Rightarrow P_2$ **where** $\phi]$, and a profitability heuristic ρ, and returns a transformed program. The $Apply$ function first uses pattern matching to find all locations in the program p where the pattern P_1 occurs. Then for each match that is found, $Apply$ evaluates the side condition ϕ to make sure that the match is valid. Our current prototype checks side conditions conservatively using read/write sets. For example, to guarantee that a statement s_1 does not modify another statement s_2, we check that $WriteSet(s_1) \cap ReadSet(s_2) = \emptyset$.

For each match that is found where the side condition holds, $Apply$ builds a substitution θ that records information about the match: θ maps the free variables in P_1 to concrete fragments of p, and it also records the location where the match occurred in p. $Apply$ collects the resulting substitutions θ into a set Θ, and then it calls the profitability heuristic ρ with Θ as a parameter. The role of the profitability heuristic ρ is to select from the set Θ of all substitutions that have been found (representing all the possible applications of the transformation rule) those substitutions that it wants to apply. Because all the substitutions in Θ represent correct transformations, it does not matter which subset the profitability heuristic chooses, and so the profitability heuristic can perform arbitrary computation without being trusted. The above approach to profitability heuristic uses the generate-and-test approach presented in the Cobalt system [14]. Alternatively, an execution engine could also employ the more demand-driven approach used in the Rhodium system [15], where side conditions directly refer to profitability facts, thus constraining which matches are explored.

function $SwPipe(p) :=$
  let $p' := Apply(p, t_1, \rho_{sw})$
  if $(p' = p)$ **then** p' **else** $SwPipe(Apply(p', t_2, \lambda x.x))$

Figure 12. Implementation of Software Pipelining using $Apply$.

Once the profitability heuristic has selected the set of substitutions it wants to apply, the $Apply$ function performs the corresponding transformations. If the profitability heuristic returns substitutions that overlap in the program fragments they match, then the $Apply$ function picks an order to apply the substitutions in, and only applies a substitution θ if no previously applied substitution has transformed elements mentioned in θ.

As an example, Figure 12 shows a function $SwPipe$ that uses $Apply$ to perform software pipelining. We use t_1 to represent the first part of software pipelining (the transformation from Figure 2), and t_2 to represent the second part (the transformation from Figure 3). The $SwPipe$ function uses $Apply$ to repeatedly apply t_1 and t_2. The software pipelining profitability heuristic ρ_{sw} is applied after t_1 has run. In our prototype, we have chosen to implement ρ_{sw} by selecting matches that reduce the number of dependencies between instructions in loop bodies. Once t_1 has fired, we need to apply t_2 on the result before doing another iteration of software pipelining. As a result, the profitability heuristic for t_2 is the identity function, which simply selects all the matches.

9. Related work

Translation Validation. Our approach is heavily inspired by the work that has been done on translation validation [20, 19, 22, 6, 11, 12]. However, unlike previous translation validation approaches, our equivalence checking algorithm addresses the challenge of reasoning about statements that are not fully specified. As a result, our approach is a *generalization* of previous translation validation techniques that allows optimizations to be proved correct once and for all. Furthermore, because our PEC approach can handle concrete statements as well as parameterized statements, it subsumes many of the previous approaches to translation validation, for example the relation approach of Necula [19] and the permute approach of Zuck *et al.* [6, 33].

Proving loop optimizations correct. Our approach to reasoning about loop reordering transformations by having a single canonical representation for all these transformations is similar to the translation validation work of Zuck *et al.* [6] and the legality check approach of Kelly et al. [9]. However, both these approaches perform runtime validation of concrete programs instead of once and for all reasoning about parameterized programs.

Automated correctness checking of optimizations. As with our PEC algorithm, the Cobalt [14] and Rhodium [15] systems are able to check the correctness of optimizations once and for all. However, Cobalt and Rhodium only support rewrite rules that transform a single statement to another statement, thus limiting the kinds of optimizations they can express and prove correct. Our PEC approach can handle complex many-to-many rewrite rules explicitly, allowing it to prove many more optimizations correct.

Human-assisted correctness checking of optimizations. A significant amount of work has been done on manually proving optimizations correct, including abstract interpretation [3, 4], the work on the VLISP compiler [7], Kleene algebra with tests [10], manual proofs of correctness for optimizations expressed in temporal

logic [26, 13], and manual proofs of correctness based on partial equivalence relations [1]. Analyses and transformations have also been proven correct mechanically, but not automatically: the soundness proof is performed with an interactive theorem prover that requires guidance from the user. For example, Young [32] has proven a code generator correct using the Boyer-Moore theorem prover enhanced with an interactive interface [8]. As another example, Cachera *et al.* [2] show how to specify static analyses and prove them correct in constructive logic using the Coq proof assistant. Via the Curry-Howard isomorphism, an implementation of the static analysis algorithm can then be extracted from the proof of correctness. Leroy's Comcert project [16] has also used a similar technique to manually develop a semantics preserving, optimizing compiler for a large subset of C. The Comcert compiler provides an end-to-end correctness guarantee, and does not just focus on optimizations, as we do in our approach. Tristan *et al.* has also proved that certain translation validators are correct once and for all, but here again by implementing the proof manually [28, 29]. In all these cases, however, the proof requires help from the user. In contrast to these approaches, our proof strategy is fully automated but trusts that the side conditions are computed correctly when the compiler executes.

Languages for expressing optimizations. The idea of analyzing optimizations written in a specialized language was introduced by Whitfield and Soffa with the Gospel language [30]. Many other frameworks and languages have been proposed for specifying dataflow analyses and transformations, including Sharlit [27], System-Z [31], languages based on regular path queries [25], and temporal logic [26, 13]. None of these approaches addresses automated correctness checking of the specified optimizations.

10. Conclusion

We developed and implemented Parameterized Equivalence Checking (PEC), a technique for automatically proving optimizations correct once and for all. PEC works by proving transformations correct on parameterized programs, thus generalizing previous translation validation techniques and adapting them to provide once and for all correctness proofs. Furthermore, our use of expressive many-to-many rewrite rules and a robust proof technique enables PEC to automatically prove correct optimizations that have been difficult or impossible to prove in previous systems.

References

[1] Nick Benton. Simple relational correctness proofs for static analyses and and program transformations. In *POPL*, 2004.

[2] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *ESOP*, 2004.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL*, 2002.

[5] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. *Journal of the Association for Computing Machinery*, 52(3):365–473, May 2005.

[6] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, May 2005.

[7] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a verified implementation of Scheme. *Lisp and Symbolic Compucation*, 8(1-2):33–110, 1995.

[8] M. Kauffmann and R.S. Boyer. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[9] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Languages and Compilers for Parallel Computing*, 1994.

[10] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Langauges and Systems*, 19(3):427–443, 1997.

[11] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Automated refinement checking of concurrent systems. In *ICCAD*, 2007.

[12] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Validating high-level synthesis. In *Computer Aided Vefification (CAV)*, 2008.

[13] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *POPL*, 2002.

[14] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, 2003.

[15] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[16] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, 2006.

[17] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[18] S. Muchnick. *Advanced Compiler Design And Implementation*. Morgan Kaufmann Publishers, 1997.

[19] G. Necula. Translation validation for an optimizing compiler. In *PLDI*, June 2000.

[20] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.

[21] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:4–13, 1992.

[22] Martin Rinard and Darko Marinov. Credible compilation. In *Proceedings of the FLoC Workshop Run-Time Result Verification*, July 1999.

[23] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: a new analysis framework for parallelizing compilers. In *PLDI*, 1996.

[24] Hanan Samet. Proving the correctness of heuristically optimized code. *Commun. ACM*, 21(7):570–582, July 1978.

[25] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *POPL*, 2004.

[26] Bernhard Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 346–364. Springer-Verlag, September 1991.

[27] Steven W. K. Tjiang and John L. Hennessy. Sharlit – a tool for building optimizers. In *PLDI*, 1992.

[28] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *POPL*, 2008.

[29] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *PLDI*, 2009.

[30] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[31] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *POPL*, 1993.

[32] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.

[33] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, 2005.