Conditionally Correct Superoptimization

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Eric Schkufza
Stanford University
eschkufz@cs.stanford.edu

Berkeley Churchill
Stanford University
bchurchill@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

The aggressive optimization of heavily used kernels is an important problem in high-performance computing. However, both general purpose compilers and highly specialized tools such as superoptimizers often do not have sufficient static knowledge of restrictions on program inputs that could be exploited to produce the very best code. For many applications, the best possible code is *conditionally correct*: the optimized kernel is equal to the code that it replaces only under certain preconditions on the kernel's inputs. The main technical challenge in producing conditionally correct optimizations is in obtaining non-trivial and useful conditions and proving conditional equivalence formally in the presence of loops. We combine abstract interpretation, decision procedures, and testing to yield a verification strategy that can address both of these problems. This approach yields a superoptimizer for x86 that in our experiments produces binaries that are often multiple times faster than those produced by production compilers.

1. Introduction

The aggressive optimization of heavily used kernels is an important problem in high-performance computing applications. However, both general purpose compilers and special purpose tools such as superoptimizers [1] [3, 22, 35] often fail to produce the best possible code. In many cases, this is because they are unaware of restrictions on a kernel's possible runtime inputs. As a result, these tools must forgo aggressive optimizations that are correct for inputs that are guaranteed to arise at runtime but may be incorrect for inputs that in fact cannot occur in the specific context in which the kernel is used.

Transformations of this form fall into the category of *conditionally correct* optimizations: the resulting code is equal to the code that it replaces only under certain preconditions on kernel inputs. Due to the significant benefits associated with these transformations, modern compilers provide some facilities for programmers to assert preconditions for important conditionally correct optimizations. For example, `gcc` provides support for a small set of annotations that can be used to communicate contextual hints to its optimization routines. The `restrict` keyword of the C99 standard, for instance, declares that a function will only be executed in contexts where addresses derived from distinct input pointers cannot overlap. Using this annotation, `gcc` is sometimes able to perform a conditionally correct optimization that results in code which is correct only in contexts that respect that annotation.

The value of conditionally correct optimizations is shown in Figure 1 (further details of this evaluation are presented in Section 5), which shows the performance improvement of code compiled with `gcc -O3` using annotations (`annot gcc -O3`) over a baseline of `gcc -O3`. For these benchmarks, the use of compiler annotations results in as much as a 3× speedup over code that is already aggressively optimized (`O3` is the highest level of optimization that `gcc` provides). However, for many benchmarks the use of annotations does not produce any improvement at all, and in some cases can even result in slowdowns (`ex3c`). Regardless, this situation is still an improvement over current superoptimizers [3, 22, 35] that provide no facilities for consuming annotations and reject optimizations that are not provably correct for all possible inputs.

There are several reasons why a compiler might be unable to take complete advantage of hints that describe constraints on execution context. First, compilers are designed to provide fast compilation times, and the static analyses that meet this criteria are often too imprecise to prove the correctness of the desired optimization even in the presence of a restricted context. As a result, the hint is often ignored and a potential performance improvement is lost. Second, the language of annotations currently supported by production compilers is quite restrictive. Many of the hints that a pro-

[1] The discussion in this section is restricted to superoptimizers that formally guarantee correctness. Also see Section 6.

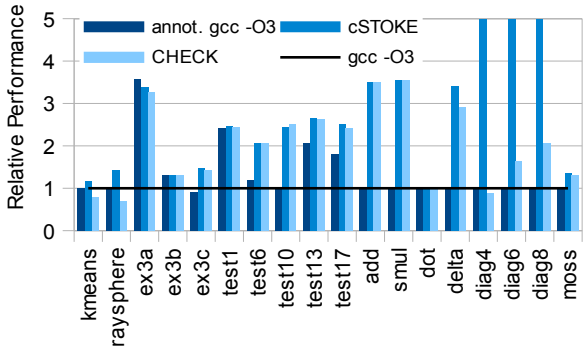

Figure 1. The value of conditionally correct optimization, `gcc` with full optimizations enabled (`-O3`) shown for reference. Vertical bars show speedups for code obtained by providing `gcc` with conditionally correct annotations (`annot gcc -O3`), the conditionally correct code generated by `cSTOKE`, and `cSTOKE` code which is instrumented to check correctness conditions at runtime (`CHECK`).

grammer would like to provide, such as ranges of program inputs, are not currently supported. This situation is slowly improving, but is still quite far from a state in which hints can be provided in full generality. And finally, for many low-level optimizations, it is not clear what hints, and in what combination, are necessary to produce faster code. The hints that a programmer considers useful might not have any, or as suggested, even an adverse effect on a compiler's ability to generate performant code. And in many cases, missed annotations may lead directly to missed optimizations.

In this paper, we present an approach to conditionally correct optimizations that overcomes these issues by inverting the annotation-based process described above. Rather than ask the user to provide annotations in the hope that they will assist the compiler in producing optimized code, we ask the compiler to produce optimized code along with the preconditions required to demonstrate the code's conditional correctness. To maintain generality, we use a set of user-provided test cases (i.e., concrete inputs) to describe possible runtime contexts. In exchange, the compiler produces both an optimized kernel and the precondition under which it is formally guaranteed to be correct. All of the test cases are guaranteed to satisfy the precondition, and if the user believes that the inferred precondition is too strong, he can supply additional test cases that cover the missing behaviors and the process can be repeated.

Because the code produced by the optimizer is only conditionally correct, it is not guaranteed to hold in an arbitrary execution context. Furthermore, because the verification technology for proving that a precondition is always satisfied in the context of real world programs does not currently exist, we take a different approach. The class of preconditions we consider are easily (and mechanically) convertible to executable code. As a result, it is straightforward to check those preconditions at runtime and then to either execute the optimized code if they hold or the original code if they do not. In some cases the cost of checking a precondition may outweigh the benefit of using the optimized code. If so, the user can either dispense with the runtime checks if he can verify that they will always be satisfied by other means, or dispense with the optimization altogether.

We demonstrate the feasibility of this approach in a tool based on STOKE [35, 36, 39], a binary superoptimizer for x86 which produces code that is correct for *all* possible input contexts. Our modified version of STOKE, called `cSTOKE` (STOKE with conditions), is capable of producing conditionally correct binaries and contains a bit-precise verifier called COVE (COnditional VErifier) that makes use of a combination of abstract interpretation, decision procedures, and testing to infer the conditions required to verify the conditional correctness of optimized code. The resulting optimizer addresses the three concerns described above: it uses state-of-the-art techniques to verify conditionally correct optimizations that production compilers cannot, it is capable of mining arbitrary information about preconditions from test cases, and it can infer the preconditions required to justify aggressive optimizations automatically.

Using this superoptimizer (`cSTOKE`, Figure 1) we are able to match, and in many cases improve, the performance of `gcc` by inferring and exploiting contextual information. Although STOKE has previously been shown to produce correct binaries that are up to 70% faster than those generated by production compilers [35, 39], by leveraging conditional correctness `cSTOKE` can generate binaries that are several times faster regardless of whether or not annotations are used. Figure 1 (`CHECK`) shows the performance of `cSTOKE` code with instrumentation that checks the inferred preconditions at runtime (Section 5.5). Although the resulting binaries are correct for all inputs, the instrumentation can cause significant performance degradation to the extent that the benefits of conditionally correct optimizations are lost (`kmeans` and `raysphere`). Nonetheless, for most benchmarks, the code obtained by instrumenting conditionally correct `cSTOKE` code is significantly better than either the correct or conditionally correct code generated by `gcc`.

We wish to stress that our contribution lies in improved verification technology rather than in the improved ability to discover optimizations. The original implementation of STOKE without support for conditional correctness [35, 39] can find all of the optimizations that we report in this paper. However, the original STOKE verifier can only prove the correctness of optimizations when the improved program is equivalent to the original for all possible inputs. As a result, it cannot verify any of the conditionally correct optimizations that we report and must instead discard them. Remedying this shortcoming is precisely what distinguishes STOKE from `cSTOKE`.

Although the benchmarks we consider are small (less than one hundred lines of 64-bit x86 assembly), they are representative of the complexity limits of STOKE, and extend well beyond the capabilities of other superoptimizers. For example, [3] (the only other superoptimizer for x86) is limited to loop-free x86 programs of six lines or less. And although the running time of our optimizer is longer than that of a traditional compiler, the benchmarks that we consider are representative of high-performance compute kernels for which the additional optimization time is justified. In general, we find that the optimization cost (seconds to minutes) is acceptable given the high quality of the resulting binaries.

To summarize, this paper makes the following contributions. In Section 2, through an example, we describe our approach to compiler optimization: an optimizer generates performant code that is correct only in restricted contexts and presents a formal description of those preconditions. In Section 3, we present an algorithm for verifying conditionally correct optimizations. The main technical contribution of this algorithm is in showing that tests can be used both to obtain useful conditions under which it is possible to prove conditional equivalence and to infer invariants over arbitrary abstract domains that relate the original and optimized programs. In Section 4, we describe the implementation of the first binary optimizer for x86 that produces conditionally correct binaries that are formally correct under the conditions also included in the output. In Section 5 we show that, in our experiments, conditionally correct code is often multiple times faster than the code generated by production compilers and maintains desired application behavior, and discuss instrumentation for checking the inferred conditions dynamically at runtime. We conclude in Section 6 with a discussion of related work.

2. Overview and Motivating Example

We begin with an informal overview of our approach to conditionally correct optimization with an example of a ray tracing program [36]. As is typical of ray tracers, the overall execution time of the program is dominated by vector arithmetic. In particular, consider the code shown in Figure 2, which executes in the inner loop of the application and induces depth-of-field blur by repeatedly perturbing the viewing camera angle. By improving the performance of this kernel, it is possible to improve overall program performance.

We first provide STOKE with the binary of a program and the name of a function to optimize. In this case, we use the code produced by compiling the ray tracer from the C source using `gcc -O3` and instruct STOKE to optimize the resulting camera perturbation code shown in Figure 2 (left).[2] STOKE runs the program on a small set of user-provided test cases and snapshots the machine states that

```
V delta(V& v1, V& v2, float r1, float r2) {
  // v1  = [(rdi),      4(rdi),      8(rdi)    ]
  // v2  = [(rsi),      4(rsi),      8(rsi)    ]
  // ret = [xmm0[63:32], xmm0[31:0], xmm1[31:0]]

  assert(0.0 <= r1 <= 1.0 && 0.0 <= r2 <= 1.0);

  // gcc -O3:
  return V(99*(v1.x*(r1-0.5))+99*(v2.x*(r2-0.5)),
           99*(v1.y*(r1-0.5))+99*(v2.y*(r2-0.5)),
           99*(v1.z*(r1-.05))+99*(v2.z*(r2-0.5)));
  // STOKE:
  return V(99*(v1.x*(r1-0.5)),
           99*(v1.y*(r1-0.5)),
           99*(v2.z*(r2-0.5)));
}
```

```
 1 # gcc -O3
 2
 3 movl 0.5, eax
 4 movd eax, xmm2
 5 subss xmm2, xmm0
 6 movss 8(rdi), xmm3
 7 subss xmm2, xmm1
 8 movss 4(rdi), xmm5
 9 movss 8(rsi), xmm2
10 movss 4(rsi), xmm6
11 mulss xmm0, xmm3
12 movl 99.0, eax
13 movd eax, xmm4
14 mulss xmm1, xmm2
15 mulss xmm0, xmm5
16 mulss xmm1, xmm6
17 mulss (rdi), xmm0
18 mulss (rsi), xmm1
19 mulss xmm4, xmm5
20 mulss xmm4, xmm6
21 mulss xmm4, xmm3
22 mulss xmm4, xmm2
23 mulss xmm4, xmm0
24 mulss xmm4, xmm1
25 addss xmm6, xmm5
26 addss xmm1, xmm0
27 movss xmm5, -20(rsp)
28 movaps xmm3, xmm1
29 addss xmm2, xmm1
30 movss xmm0, -24(rsp)
31 movq -24(rsp), xmm0
```

```
 1 # STOKE
 2
 3 movl 0.5 eax
 4 movd eax, xmm2
 5 subps xmm2, xmm0
 6 movl 99.0, eax
 7 subps xmm2, xmm1
 8 movd eax, xmm4
 9 mulss 8(rsi), xmm1
10 movss 4(rdi), xmm5
11 mulss xmm0, xmm5
12 mulss (rdi), xmm0
13 mulss xmm4, xmm0
14 mulps xmm4, xmm5
15 punpckldq xmm5, xmm0
16 mulss xmm4, xmm1
```

Figure 2. A conditionally correct optimization for a routine that generates random camera perturbations (top). STOKE produces code that is faster than `gcc -O3` (bottom left) by eliminating and reordering computations. The resulting code (bottom right) is proved conditionally correct using COVE.

immediately precede the execution of the perturbation code. These machine states are used as test cases to check whether putative optimizations preserve the function's behavior on at least some inputs. The code generated by `gcc` reads from six heap locations and writes to two stack locations. However, the contents of some heap locations depend only on the position of the camera and remain constant irrespective of

[2] The instructions in this code are in the AT&T syntax, i.e., they follow an opcode-source-destination pattern. E.g., the instruction `movl ebx, eax` moves the contents of register `ebx` to `eax`. Arithmetic instructions consider the destination as the first operand. E.g., the instruction `subl 1, eax` decrements the integer represented by the bits in `eax` by 1.

the scene being rendered. As a result, STOKE is able to generate the shorter code shown in Figure 2 (right) with the guarantee that it agrees with the `gcc` code on every provided test case. The resulting code both omits reads from locations that hold constant values and avoids the use of the stack by exploiting vector instructions.

Having produced a potential optimization, STOKE attempts to use its formal verifier to prove the equivalence of the two programs shown in Figure 2 for all possible inputs; as expected, the verification fails. In the standard formulation of compiler optimization, STOKE has proposed an incorrect transformation: although the faster program is correct for every user-provided test case, it would be incorrect were it run with different values in the heap. However, the optimization performed by STOKE is in fact conditionally correct under conditions that hold for the ray tracer. As a result, we apply a subsequent check for conditional equivalence that, if successful, presents the user with both the optimization and the conditions under which it holds.

The approach so far described is similar to speculative optimizers (including profile guided optimizers and JIT compilers) that apply conditionally correct optimizations and must produce the conditions under which they are correct (in order to enforce them at runtime). However, there is a major difference between these techniques and our approach. For speculative optimizers, the optimizer and the verifier are intertwined; the optimizer has a list of transformations and the conditions under which it is safe to apply them. As a result, obtaining the conditions under which the optimized code is correct is direct and results from conjoining the conditions for every applied transformation.

In contrast, the non-traditional optimizations produced by superoptimizers that enumerate arbitrary programs make condition inference substantially more difficult. STOKE makes random changes to the input program until it finds a code that both agrees with that program on all test cases and produces better performance (Section 4.1). The random changes are neither required to preserve correctness nor improve performance. In a typical run STOKE makes millions to billions of random changes and often finds optimizations outside the vocabulary of a traditional optimizer. This characteristic of producing surprising code sequences is what makes superoptimization so powerful. In contrast to a speculative optimizer that has perfect knowledge about the optimizations that have been applied and the conditions under which they are correct, with STOKE we have no such knowledge. The output of STOKE is an arbitrary sequence of x86 instructions that agrees with the input program on all the tests. Beyond that, we have no useful information regarding what program changes have been applied to transform the input program to the output program.

Before discussing our implementation of conditional equivalence for such code sequences, we note that the two programs shown in Figure 2 are conditionally equivalent under several possible conditions, many of which are useless. First, for example, any two programs are equivalent under the condition *false*. Second, the two programs are equivalent under the condition that encodes the union of the available test cases; all outputs of STOKE satisfy this condition. Thirdly, the weakest precondition is a direct logical encoding of the two x86 programs and provides the trivial guarantee that the programs are equivalent on all inputs for which the programs produce the same output.

To be useful, a set of conditions must provide non-trivial guarantees. COVE first automatically computes a sound over-approximation of the user-provided test cases to infer non-trivial human-comprehensible preconditions and then proves conditional equivalence under those conditions. For the camera perturbation code, COVE produces conditions that state that the values at memory locations `8(rdi)`, `(rsi)`, and `4(rsi)` are zero, and that the memory instructions on lines 6, 8, 9, 10, 16, 17, and 18 must read from distinct memory locations.

COVE then attempts to prove the equivalence of the optimized program under these conditions. For both the original and optimized programs, COVE translates the x86 code to SMT formulae that soundly model execution with bit-precise accuracy. Crucially, because COVE is designed to prove conditional equivalence, these formulas can be much more compact than formulas that soundly model execution for the purpose of proving equivalence under all possible contexts. For example, to prove unconditional equivalence we must generate constraints for both an aliasing and non-aliasing case for every pair of memory dereferences. These cases grow exponentially in the number of memory dereferences. In contrast, to prove conditional equivalence, the constraints need only model the aliasing relationships that are inferred from the user-provided tests. Because these inferred aliasing relationships are included in the computed precondition (Section 3.6), these constraints model all possible executions under that precondition. The consequences are dramatic. For the example in Figure 2, the reduction in the number of cases is approximately nine orders of magnitude! As a result, the constraints are within the reach of state of the art SMT solvers. COVE creates a formula whose satisfying assignments are initial program states that satisfy the preconditions and cause the two programs to produce different results. If the SMT solver returns "unsat", COVE has proven conditional equivalence. Although the resulting SMT queries are hard—they contain quantifiers, intermix bit-vectors ranging from 8 bits to 128 bits, and the constraints that model the x86 instructions are quite complex—the verification time for all of the benchmarks we consider is well under one second.

Having produced both an optimization and the precondition under which it is correct, the user must decide (based on the inferred preconditions) whether more tests are needed to cover missing relevant behavior. In addition, the user has the

option of either instrumenting the code with runtime checks for those preconditions (and in doing so guaranteeing correctness) or running the code as is. For the code shown in Figure 2 (right) this results in over 3× speedup (slightly under 3× if dynamic checks are included) compared to the code generated by `gcc -O3`. When combined with optimizations to three additional kernels (Section 5), the end-to-end speedup over the original ray tracer application is over 2× for rendering a full scene.

3. Verification

We present a formal definition of conditionally correct optimizations, a generator of useful conditions, and a verification strategy for proving conditional equivalence. We start by defining conditional equivalence in the context of compiler optimizations.

3.1 Conditional Equivalence

We make a distinction between T, the *target* or reference code, and R, the *rewrite* or proposed optimized replacement for the target T. A program *state* consists of a valuation of registers and memory. The 64-bit x86 architecture has sixteen 64-bit general purpose registers, sixteen 128-bit SSE registers, and memory, which we model as an array. When we refer to the state at a program point, that state is limited to live registers and memory locations.

Two programs are conditionally equivalent if they are equivalent for all input states that satisfy a given condition.

Definition 1. *Target T is conditionally equivalent to rewrite R under the condition C if for all states s satisfying $C(s)$ both of the following hold: (i) if executing T from initial state s terminates in state s' without aborting, then executing R from initial state s also terminates in state s' without aborting; and (ii) if T diverges when execution is started from s, then so does R.*

This definition captures the requirement that T terminates on an input s if and only if R terminates on s. Hence, this definition is richer than partial equivalence. The asymmetric notion of equality with respect to aborting in Definition 1 seems necessary to validate several useful compiler optimizations. If the target T aborts (generates a hardware fault) on some input, optimizing compilers are free to use an R with any behavior, defined or undefined, on that input.

An optimization is *conditionally correct* if it produces conditionally equivalent rewrites. An optimization is (unconditionally) *correct* if it produces rewrites that are conditionally equivalent with condition *true*. Our framework COVE takes two programs (T, R) and a set S of tests, proves their conditional equivalence, and produces the condition C as output.

3.2 Verification Conditions

In this section we start with a simple example for the task of conditional equivalence checking, but one that is sufficient to illustrate the important concepts. Consider the two programs T and R of Figure 3 which manipulate a global variable `x`. A sufficient condition for T and R to be conditionally equivalent is the existence of an invariant I and a condition Φ that make the four *verification conditions* (VCs) of Figure 3 valid. Note that all the variables in all the VCs are implicitly universally quantified.

```
1 void T() {
2   x = 0;
3   while (x < 10){
4     x = x+1;
5 }
```

```
1 void R() {
2   while(x!=10){
3     x=x+1;
4}
```

$$\Phi(x) \wedge x = x_1 = x_2 \wedge x_1' = 0 \wedge x_2' = x_2 \Rightarrow I(x_1', x_2') \quad (Init)$$

$$I(x_1, x_2) \wedge x_1 < 10 \wedge x_1' = x_1 + 1 \wedge x_2 \neq 10 \wedge x_2' = x_2 + 1 \Rightarrow I(x_1', x_2') \quad (Ind)$$

$$I(x_1, x_2,) \wedge x_1 \geq 10 \ \wedge x_2 = 10 \Rightarrow x_1 = x_2 \quad (Partial)$$

$$I(x_1, x_2) \Rightarrow (x_1 < 10 \Leftrightarrow x_2 \neq 10) \quad (Total)$$

Figure 3. Two conditionally equivalent loops and the corresponding VCs. The variables x_1 and x_2 represent the states of T and R, respectively. COVE infers the condition Φ and the invariant I that make these VCs valid.

The condition Φ constrains the input contexts in which T and R start their execution. The first VC *Init* states that when we start execution of T and R from equivalent states that satisfy Φ and the execution reaches the loop heads then the states of T and R are related by the invariant I. We call this VC *Init* as it constrains the invariant to hold initially before the loops execute. The second VC *Ind* states that if we start the execution of the loop bodies of T and R in states that satisfy I and execute one iteration of the loops then the resulting states of T and R are again related by I. We call this VC *Ind* as it constrains the invariant to be inductive. Any predicate that satisfies *Init* and *Ind* is an *inductive invariant*: it both holds initially and is inductive. However, not all inductive invariants are strong enough to prove equivalence. For example, *true* is a trivial inductive invariant. The third VC *Partial* says that I is strong enough to prove partial equivalence (i.e., equivalence modulo termination). It states that if T exits the loop and R exits the loop then the final states of T and R are equivalent. The last VC *Total* is needed to prove (total) equivalence as given by Definition 1. It says that T exits the loop if and only if R exits the loop. For the interested reader, we explain the VCs in detail in Appendix A.

Given this formulation, we have reduced the problem of checking conditional equivalence between T and R to finding an unknown invariant I and a condition Φ that make the given VCs valid. We follow the approach described in DDEC [39] to generate the VCs with one major difference that we discuss in Section 3.6. DDEC assumes System V AMD64 ABI as the calling convention [28] and includes a dataflow analysis for liveness. We refer the reader to [39]

COVE(V: VCs, S: Concrete States, $\mathcal{A}$: Abstract domains)
Returns: A condition Φ that makes V valid

```
1:  Φ := true; I := true
2:  for each (A, α, ⊔) ∈ 𝒜 do
3:    if Valid(V, Φ, I) then
4:      RETURN Φ
5:    end if
6:    Φ:= Φ ∧ α(S.PreTests)
7:    J := α(S.InvTests); Δ := ∅
8:    repeat
9:      J := J ⊔ α(Δ)
10:     Δ := Check(V.Init, V.Ind, Φ, I ∧ J)
11:   until Δ = ∅
12:   I := I ∧ J
13: end for
14: if Valid(V, Φ, I) then
15:   RETURN Φ
16: else
17:   RETURN false
18: end if
```

Figure 4. The COVE algorithm for proving the conditional equivalence of two loops. COVE combines testing (lines 6 and 7), decision procedures (lines 3, 10 and 14), and abstract interpretation (lines 8 to 11).

for a detailed description of VC generation for x86. The VC generation is straightforward if the loops run for equal numbers of iterations and the resulting VCs are analogous to those shown in Figure 3: there is one VC for the first execution of the two loop heads, one for establishing inductiveness, one for partial equivalence, and one for total equivalence. More complicated control flow, for example nested loops, require additional VCs and invariants. There are standard techniques to generate VCs for unstructured programs [4, 12] and the x86 specific details can be found in [39]. Every additional loop requires an additional invariant and the number of VCs grows linearly with the number of loops (nested or otherwise). For all of our benchmarks, the loops have identical control flow and run for the same number of iterations. If the loops run for different numbers of iterations then [39] uses tests to infer how the numbers of loop iterations are related and performs loop unwinding and loop unrolling to produce two loops that run for equal numbers of iterations on the tests and then generates the VCs. This approach fails to generate VCs to prove the equivalence of very different programs, say the equivalence of two conceptually distinct algorithms, but we find it to be effective for proving equivalences relevant to optimizations.

3.3 COVE

The core COVE algorithm takes a set of VCs, V, a set of concrete states, S, and a set of abstract domains, $\mathcal{A}$. The VCs contain an unknown precondition Φ and an unknown invariant I. COVE finds a valuation of these unknown predicates that makes the VCs valid. The algorithm can be generalized to VCs with multiple unknown invariants and hence to more complicated programs [38, 39]. The pseudocode is depicted in Figure 4; we explain the details next.

The input S contains the concrete states that are inputs to T ($S.PreTests$) and the concrete states observed on each iteration of the loops ($S.InvTests$). We choose $S.PreTests$ to be a set of input states on which T and R are equivalent. For each $s \in S.PreTests$ we then run T and R on s and record the set of states encountered at the loop heads. We place all states encountered at the loop head in $S.InvTests$.

Each abstract domain $A \in \mathcal{A}$ is ordered by $\sqsubseteq$, has an abstraction function α, a join function $\sqcup$, and if needed a widening operator ∇. The conditions and invariants inferred by COVE are restricted to abstract states in A. We assume that each abstract value can be converted to a predicate that can be consumed by a decision procedure.

The algorithm starts by initializing Φ and I to *true*. Next, we iterate over each abstract domain to strengthen these conditions. If the current invariant and the condition make the VCs valid then we return the condition and exit (lines 3-5 and 14-15). Otherwise, we update the condition Φ by conjoining it with $\alpha(S.PreTests)$ and initialize the candidate invariant J over the current abstract domain to $\alpha(S.InvTests)$ (lines 6 and 7). We then iteratively use the *Check* decision procedure to find any counterexamples which show that J is not an inductive invariant. If any counterexamples Δ are found then we join J with $\alpha(\Delta)$ and repeat (as [33, 38, 39, 44]). Otherwise, if the check succeeds (no counterexamples are returned) then J is an inductive invariant and the current invariant I is strengthened. For some abstract domains, such as intervals, the joins need to be replaced with widening to guarantee termination. After updating I, we repeat this process with the next abstract domain in $\mathcal{A}$. If none of the abstract domains suffice then COVE fails.

3.3.1 Example

We show how COVE proves the conditional equivalence of the programs in Figure 3. Choose $S.PreTests$ to be $\{x = 0\}$ since on this input set, T and R are equivalent. Then we find $S.InvTests \equiv \{(x_1 = 0, x_2 = 0), \ldots, (x_1 = 10, x_2 = 10)\}$ (where x_1 denotes the state of T and x_2 denotes the state of R). Suppose we first consider the domain of affine equalities [23]. Using S as described, line 6 sets Φ to $x = 0$ and line 7 sets J to $x_1 = x_2$. A decision procedure is able to check that this instantiation makes the predicates $\neg Init$ and $\neg Ind$ unsatisfiable in line 10. Hence, line 12 updates the current invariant I to $x_1 = x_2$. However, I is not strong enough to make all the VCs valid on line 3. We repeat this exercise with intervals [8]. We have $\alpha(S.PreTests) \equiv x = 0$ and $\alpha(S.InvTests) \equiv (0 \leq x_1 \leq 10) \wedge (0 \leq x_2 \leq 10)$. Again the check on line 10 succeeds (no counterexamples are found) and I is strengthened to $(x_1 = x_2) \wedge (0 \leq x_1 \leq 10)$. This time the check on line 3 succeeds and $x = 0$ is returned as the condition under which T and R are equivalent. Note that any one of equalities or intervals is sufficient to ex-

press the inferred condition. However, they individually are insufficient to prove conditional equivalence and the proof requires invariants over both abstract domains. In our evaluation, we find that it is often the case that an abstract domain is absent in conditions but it is still necessary to express the requisite invariants (Section 5.3).

3.4 Properties of COVE

The standard techniques to infer preconditions in program verification attempt to perform a weakest precondition computation. However, there is no general algorithm to compute weakest preconditions in the presence of loops. For loop-free programs, the weakest precondition computation is intractable in the presence of bit-vector operations and aliasing [2, 6]. It is also not clear how to approximate the weakest precondition while not ruling out the given valid executions. The condition $\Phi \equiv \mathit{false}$ satisfies the VCs but yields a trivial guarantee: two programs are equivalent if they are never executed. Our choice for Φ captures information in the test cases under which T is executed.

The condition discovered in Figure 4 is the strongest abstraction of the test cases in the given abstract domain. The following lemma shows that, for a given iteration of the outer loop, we can ensure that the inductive invariant J found by the inner loop is the most precise inductive invariant that can be expressed in our abstract domain. We omit the proof as it uses standard techniques [33, 38].

Lemma 3.1. *For all inductive invariants $\mathcal{I} \in A$, the candidate invariant $J \in A$ satisfies $J \sqsubseteq \mathcal{I}$.*

If the checks on lines 3 or 14 of Figure 4 fails then either the VCs model the concrete semantics too imprecisely or the abstract domain is insufficient and COVE must be re-executed with another abstract domain. Lemma 3.1, which intuitively says that COVE does not suffer from excess over-approximation, shows that there can be no other source of failure.

Even with this per-iteration guarantee, it is useful to have a weakening phase before returning the precondition, as the obtained condition might be unnecessarily strong. In the weakening phase, we find the minimal condition under which we are able to prove equivalence. Since our conditions are small, we currently perform a linear scan over the conjuncts in the inferred condition to find a minimal condition. For example, for the condition $x \geq 0 \wedge x \leq 0$, we can try dropping the conjuncts one by one. For Figure 3 none of these can be omitted.

In the absence of tests, our iterative procedure can still be used by initializing the candidate invariant with *false* (instead of $\alpha(S)$). However, in this case the convergence to an inductive invariant may be slow. In particular, for intervals, the iterative process with no tests requires ten iterations to find an inductive invariant for Figure 3. By design, COVE has access to tests, which accelerates convergence of the iterative process. As remarked in Section 2, we focus on compute intensive kernels. Almost by definition, these kernels are executed very frequently and hence many test cases are available (up to millions, in our experience).

3.4.1 Instantiating COVE for x86

For x86 binaries, we use the following abstract domains: the alignment domain described in Section 3.5, the domain of bit-vector equalities [10], and the domain of bit-vector intervals [32]. Additional abstract domains can easily be included (e.g., relational abstract domains such as polyhedra) but so far we have not found them to be useful in our setting.

The domain of bit-vector intervals is very similar to integer intervals [8]. We define a linear ordering $\leq_b$ between bit-strings with identical bit-widths as follows: $b_1 \leq_b b_2$ if the unsigned integer represented by bits of b_1 is smaller than the unsigned integer represented by the bits of b_2. The abstractions are intervals, $[b_l, b_u]$, and the concretization $\gamma([b_l, b_u]) = \{b : b_l \leq_b b \leq b_u\}$. The abstraction function over a single concrete state b is given by $\alpha(b) = [b, b]$. Next, the join operator is defined as follows: $[c, d] \sqcup [e, f] = [min(c, e), max(d, f)]$, where the *min* and *max* operations are in accordance with the ordering $\leq_b$. Since this abstract domain has an exponential height, we define a widening operator to accelerate convergence: $[c, d] \nabla [e, f] = [g, h]$, where $[g, h]$ is the smallest interval that contains $[c, d] \sqcup [e, f]$ with the restrictions that the bit-strings g and h both have at most one bit as 1 and the rest of the bits 0.

In contrast to intervals, the domain of bit-vector equalities is more sophisticated and requires operations over matrices whose entries are bit-vectors. We arrange the concrete states in a matrix M where each row represents a concrete state and each column represents a particular state element (such as a register). The abstraction $\alpha(M)$ is the Howell normal form of M [20]. This normal form is an extension of reduced row-echelon form [19] albeit suitable for matrices with bit-vector entries and its computation incurs the same overall cubic-time complexity. The join, $M \sqcup N$ is the Howell normal form of the matrix obtained by concatenating M and N vertically. We did not require a widening operator for this domain because convergence is fast in practice. We refer the reader to [10] for a detailed exposition.

In our initial implementation of COVE, one important source of failures was floating-point instructions. Since the current support for floating-point in SMT solvers is immature, COVE models floating point operations as uninterpreted functions. For floating-point programs, a proof can fail because this modeling can be too imprecise. COVE addresses this imprecision by constraining these uninterpreted functions with universally quantified axioms and using these axioms during calls to its SMT solver (lines 3, 10 and 14 of Figure 4). Some of these axioms are unsafe and are included in the reported condition if they are used. For example, associativity of floating-point addition is an unsafe axiom, whereas commutativity is a safe axiom. Readers familiar with deci-

$$
\begin{array}{lrcl}
\textit{Condition} & C & := & \Phi \wedge P \wedge U \\
\textit{AbstractPre} & \Phi & := & \Phi_{Align} \wedge \Phi_A \wedge \Phi_B \\
\textit{Aliasing} & P & := & \textit{restrict } X \mid P \wedge \textit{restrict } X \\
\textit{UnsafeAxioms} & U & := & \ldots
\end{array}
$$

Figure 5. The grammar of inferred conditions. The conditions shown to the user consist of aliasing relationships (P), alignment restrictions (Φ_{Align}), equality relationships (Φ_A), inequality relationships (Φ_B), and unsafe floating-point axioms (U).

sion procedure implementations may wonder whether trading immature floating-point reasoning for uncertain quantifier reasoning is worthwhile, but in our experience current SMT solvers are much better at handling quantifiers than floating-point arithmetic.

The condition C shown to the user belongs to the grammar in Figure 5. Here C consists of the condition Φ generated by abstraction, the aliasing relationships (Section 3.6) composed of *restrict* on input pointers X, and the unsafe floating-point axioms U that are required by the SMT solver. The condition Φ is composed of alignment restrictions Φ_{Align}, bit-vector equalities Φ_A, and bit-vector intervals Φ_B. A verification task need not require all possible constituents. An example of a condition expressible by the language in Figure 5 is that in all tests $x = y$, floating-point multiplication of x with zero yields zero, the address in p is 16-byte aligned, the address in q is 8-byte aligned, and the addresses obtained from p and q never overlap.

3.5 Alignment

Many x86 vector instructions require the input memory addresses to be properly aligned. Validating rewrites that contain such instructions requires COVE to precisely reason about alignment. We describe the abstract domain used by COVE to infer alignment information. Consider the lattice L with the following order:

$$\bot \sqsubseteq b64 \sqsubseteq b32 \sqsubseteq b16 \sqsubseteq b8 \sqsubseteq b4 \sqsubseteq b2 \sqsubseteq b1$$

The top value $b1$ represents one byte alignment. Since x86 is byte addressable, $b1$ is the top element of the lattice. An abstract value bY represents Y byte alignment. For the current x86 ISA, alignment beyond $b64$ is irrelevant. The order represents stricter alignments: e.g., all pointers are byte aligned and if some pointer is 16 byte aligned then it is also 8 byte aligned. The join operator is simple: $c \sqcup d = max(c, d)$. Since this lattice is linearly ordered, the *max* operator is well defined. The abstraction function α when applied to a single pointer maps the address to its alignment, and α is generalized to sets of addresses in the obvious way, $\alpha(S) = \bigsqcup_{s \in S} \alpha(s)$. This abstract domain helps validate many optimizations.

As an example, consider using this domain to infer conditions for checking the equivalence of the programs shown

```
1 # T                        1 # R
2                            2
3 movlps (rax), xmm1         3 addps (rax), xmm0
4 movhps 8(rax), xmm1
5 addps xmm1, xmm0
```

Figure 6. A target program `T` along with an optimized rewrite `R`. The two programs are equivalent if the address in `rax` is 16 byte aligned.

in Figure 6. Here R loads 128 bits from the address contained in the register `rax`, treats them as four 32-bit floats a_1, a_2, a_3, a_4, treats 128 bits of register `xmm0` as four floats b_1, b_2, b_3, b_4, and then stores in register `xmm0` the four floating-point values $a_1 +_f b_1, \ldots, a_4 +_f b_4$, where $+_f$ is floating-point addition. T achieves the same computation using three instructions. However, it is not always safe to replace T by R as R aborts when the address in `rax` is not 16 byte aligned. If the addresses in all tests are 16-byte aligned then COVE would generate $\Phi_{Align} \equiv b16$ (Figure 5) as the condition and a decision procedure can then prove conditional equivalence.

Finally, the example in Figure 6 is loop free. For programs with loops, COVE also finds invariants over the alignment domain to prove that the alignment is maintained by the loop. These invariants are required to prove conditional correctness when the rewrites use instructions with alignment restrictions in the loop body.

3.6 Aliasing

COVE performs VC generation using the DDEC algorithm which is currently the only sound equivalence checker for x86 [39]. The evaluation described in [39] indicates that DDEC can take more than an hour to validate whether a given set of invariants satisfy all the VCs for programs containing fewer than ten lines of 64-bit x86 assembly and three memory dereferences. The reason for this inordinate time requirement is that the VCs generated by DDEC are huge. As discussed in Section 2, because DDEC attempts to prove equivalence for all possible contexts, the VCs must model executions under all possible aliasing configurations. This problem is further exacerbated by the fact that x86 is byte addressable and dereferences with multiple bytes can partially overlap. COVE mines the more restricted set of of aliasing relationships P (Figure 5) from tests. COVE observes these test executions and assigns as many *restrict* labels to the input pointers as is consistent with the tests. These aliasing relationships P are included in the output condition.

Next, COVE generates VCs that are specialized to the aliasing assumptions specified by P. As a result the constraints generated by COVE are much more compact. For our largest benchmark, the specialized constraints reduce the number of aliasing configurations to consider by 30 orders of magnitude over DDEC. In the best case, when no memory addresses overlap, the constraints generated by COVE are linear in program size. Because in practice loops seldom

cSTOKE(T: Target, H: Inputs)
Returns: A condition C and conditionally correct rewrite R

1: R := STOKE(T, H)
2: P := *Aliasing*(T, H)
3: V := *VCGen*(T, R, P)
4: S := *RunTests*(T, H,V) ++ *RunTests*(R, H, V)
5: Φ := COVE(V, S, $[L, \ldots]$)
6: RETURN ($P \wedge \Phi$,R)

Figure 7. The binary optimizer described in Section 4: STOKE generated rewrites are verified by COVE.

create additional aliasing [15], the constraints generated for conditional equivalence are much more concise than those generated for equivalence. On most of the benchmarks in Figure 1, the huge constraints due to aliasing cause DDEC to fail even when proving the equivalence of the target against itself. Hence, STOKE is unable to discover any equivalent rewrites for such kernels and fails to achieve any improvements. To summarize, relaxing equivalence to conditional equivalence not only leads to better optimizations (Section 2 and Section 5) but also leads to substantially more tractable verification tasks.

4. Implementation

We describe an implementation of a binary optimizer that generates conditionally correct binaries. We are given an input program T, along with tests H and the goal is to find another program R that is better than T by some metric (which can be performance, code size, power, etc.) and also conditionally equivalent to T under a condition C. In this paper, our metric is performance.

The overall architecture of our binary optimizer is shown in Figure 7. Our tool takes as input the program to be optimized T and the inputs H. It runs STOKE (Section 4.1) on T and produces a binary R that agrees with T on all tests in H. Next, it mines the aliasing relationships P (Section 3.6), which are used to obtain the VCs V (Section 3.6). After obtaining the concrete states S (by running T and R on tests in H) which the unknown predicates in V must satisfy, COVE (Figure 4) is called with V, S, and a list of abstract domains (Section 3.3). The output Φ of COVE is conjoined with P to obtain the output condition.

We describe STOKE briefly. A complete treatment of STOKE is beyond the scope of this paper, and the reader is referred to [35, 36, 39] for more information.

4.1 STOKE

STOKE [35] is a binary optimizer that separates optimization and validation. STOKE consumes a program T and tests H and makes repeated randomly selected changes to T to produce a faster program R that is correct for the tests. The random program changes are neither required to maintain correctness on the tests nor are required to improve performance. The random changes include transformations such as replacing a randomly chosen instruction with a new randomly generated instruction, or swapping two randomly chosen instructions, etc. Once STOKE has performed millions and sometimes even billions of random changes, it asks a validator [39] to prove the equivalence of the resulting binary R and T. Unlike traditional compilers or exhaustive enumeration based superoptimizers, the transformations produced by STOKE are guided by a *cost function* defined on test cases:

$$c(R;T) = \text{eq}(R;T) + \text{perf}(R;T)$$

The notation $f(x;y)$ is read "f is a function that takes x as an argument and is *parameterized* (that is, defined in terms of) y". The eq$(\cdot)$ term assigns a higher cost to rewrites R that are "further" from being functionally equivalent to T. The perf$(\cdot)$ term is a measure of the performance difference between R and T. STOKE searches for rewrites that minimize the cost by making a random change to the current rewrite R to obtain a new rewrite R'. If the cost of R' is lower than R then R' becomes the current rewrite and the process is repeated again. If instead the cost of R' is greater than the cost of R, then with some probability we still update the current rewrite to R'. This probability exponentially decays with the difference of cost between R and R' so that better rewrites are chosen more often. STOKE runs for a fixed time budget and outputs the lowest cost rewrite it finds that agrees with T on all the given tests. This rewrite is subsequently verified formally.

Randomized sampling techniques are generally effective only if they are able to maintain a high throughput rate of proposals. STOKE addresses this issue by constructing cost functions that are very efficient to evaluate. For evaluating eq$(\cdot)$ it checks the correctness on the given tests. For perf$(\cdot)$ it sums the average latencies of the instructions. In a typical run, STOKE is able to try hundreds of thousands of rewrites per second.

The benchmarks that we consider contain a maximum of one hundred lines of x86 assembly and are representative of the length to which STOKE currently scales. Although this scale is much smaller than what is expected from a compiler, it is one to two orders of magnitude larger than what is reported by other superoptimization techniques [3, 22, 27]. The scalability of STOKE is limited as it explores arbitrary x86 programs. With more than 2000 different instruction variants, the number of possible x86 programs grows very quickly with length. However, STOKE's current abilities are sufficient to include many interesting kernels [35].

In [39], STOKE uses DDEC to prove the correctness of the optimizations by proving equivalence. DDEC [39] can be seen as a specific instantiation of COVE where the inferred conditions are fixed to *true* and the abstract domain is fixed to affine equalities. DDEC rejects the optimized rewrites produced by STOKE for almost all of our benchmarks (Sec-

tion 5). As we shall see, one of the principal culprits in the failure to prove equivalence is alignment restrictions required by the rewrite but not needed in the target. In [39], DDEC is evaluated only on programs that consist exclusively of fixed-point instructions, which have relatively few alignment restrictions in comparison to floating-point operations. The generation of efficient code for our benchmarks (Section 5), most of which involve floating-point computations, often requires the use of instructions with alignment restrictions.

In our binary optimizer, we replace DDEC with COVE as the validator for STOKE optimizations and prove the conditional equivalence according to Definition 1 using the same tests that STOKE uses for optimization. Because COVE is able to reason about the contexts in which the optimizations are performed, the result is an effective binary optimizer that uses STOKE for optimizations and provides formal guarantees of correctness.

5. Evaluation

We evaluate the implementation of our binary optimizer on a number of benchmarks that are representative of `gcc` optimizations [30], the standard compute kernels that are used by researchers to compare compilers [41], real world applications such as a ray tracer [36] and MOSS, a document fingerprinting system [37]. A more detailed description of all the benchmarks is given in Section 5.4. For each benchmark we demonstrate both the ability to produce code that is high-performance and to provide the user with useful correctness preconditions.

5.1 Setup

All experiments (performance benchmarking and verification) were run on a machine with a 3.4 GHz Intel i7-4770 processor and 16GB of physical memory. For comparisons, we use the compilers `gcc-4.8` and Intel's proprietary compiler `icc-14.0`.

The kernels can be divided into two categories: those that are embedded in applications and standalone kernels from the literature. Our binary optimizer requires tests to search for a rewrite, infer the conditions and the invariants, and evaluate the performance benefits of the optimizations. For each of the stand-alone compute kernels, these tests are written manually. For the kernels that are embedded in applications, we use STOKE's built-in support for automatically extracting test cases in the course of normal program execution. The applications manufacture inputs to the kernels by calling random number generators. Hence, it is possible to generate many tests by simply running the applications multiple times. STOKE has a PinTool [26] that gathers program states reaching the kernel during the executions of the applications (see Section 4.3 of [39]). Since kernels are the most frequently executed part of the application, they have excellent coverage and we obtain many tests.

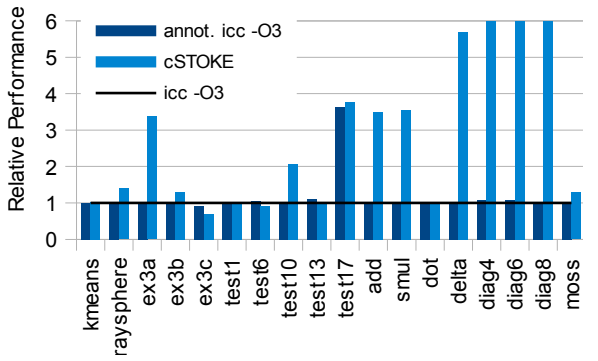

Figure 8. Performance improvement relative to `icc -O3` for `icc` using correctness preconditions discovered by COVE, and `cSTOKE`.

Next, we select a subset of these tests for STOKE and COVE (H in Figure 7). We randomly selected 32 tests from a uniform distribution over those we generated for both STOKE and COVE to use. The performance benefits are evaluated over remaining tests. We observe that a small number of tests are both sufficient to obtain high performance rewrites that are also correct under reasonable conditions. The primary reason for this result is that the loop bodies of the kernels we consider contain only a few paths (at the most 16). Although this scale is small, it is a significant improvement over previous superoptimizers which are all evaluated on straight line assembly [3, 22, 27]. Nonetheless, for more complicated benchmarks better test generation techniques might be required [13].

STOKE's search is run for 15 minutes and the best obtained rewrite is selected. In many cases, the best rewrite is found in the first few seconds. To verify the rewrite, COVE needs to query an SMT solver (Figure 4). Our implementation of COVE uses CVC4 [5] which is able to evaluate all of the SMT queries produced in the course of our experiments in under one second. Based on these results, we believe that the current generation of SMT solvers are sufficiently powerful to validate conditional correctness for many of the optimizations that are currently performed on real-world kernels.

5.2 Performance Results

A natural strategy for constructing a conditionally correct optimizer is to combine COVE with a production compiler by asserting the correctness preconditions discovered by the former using the built-in support for annotations provided by the latter. Below, we demonstrate that not only does our binary optimizer outperform `gcc` and `icc` in isolation, but also when they have been provided with the preconditions discovered by COVE.

Figure 1 uses `gcc -O3`, the most aggressive optimization level provided by `gcc`, as a performance baseline for each of the benchmarks that we consider. The verti-

cal bars labeled cSTOKE show the performance improvement that we are able to obtain by running STOKE and then using COVE to verify the conditional correctness of those optimizations. To simulate a conditionally correct implementation of gcc, we provide gcc with as much information about the inferred correctness preconditions as possible. Where appropriate, we annotate kernels with the restrict keyword (to indicate that pointers do not alias) and the set of __built_in intrinsic utilities (for instance __built_in_assume_aligned for alignment). Wherever COVE used an unsafe floating-point axiom as part of a proof, we provide the -ffast-math flag, which enables gcc to use a superset of the axioms that are available to COVE. With the exception of equalities and inequalities, the conditions can all be provided as annotations or compiler flags using the built-in support provided by gcc. The resulting performance improvement is shown in the vertical bars labeled annot gcc -O3. Although for several benchmarks the use of preconditions results in faster code, for ex3c the annotated code is slower, and overall, the performance improvements are well below those produced using STOKE.

Figure 8 compares STOKE against icc. The most aggressive optimization level provided by icc (O3) is shown as the performance baseline, and the performance improvement obtained by providing icc with the preconditions discovered by COVE is shown in the vertical bars labeled annot icc -O3. The vertical bars labeled cSTOKE are identical to those shown in Figure 1, only scaled to a different axis. As with gcc, we observe that the use of annotations often helps produce superior code. For three benchmarks STOKE fails to produce faster code than icc. For these benchmarks the code produced by icc relies heavily on software pipelining. STOKE currently uses a relatively simple performance metric in its cost function (recall Section 4.1) that ignores behavioral effects that cross loop boundaries, and as a result STOKE's code is worse than icc's code on some benchmarks. Moreover, since icc generated code is better than gcc generated code for most (but not all) benchmarks, there is less room for improvement and the speedups are lower in comparison to Figure 1. Nonetheless, we observe that for eleven out of eighteen benchmarks the code produced by STOKE is significantly more performant than the icc generated alternative, either with or without annotations.

Finally, for almost all benchmarks, STOKE as described in previous work [35, 39] is unable to achieve any improvement over the baseline. Although the optimizations that it produces are correct, it is unable to verify their correctness primarily due to the exponentially many cases that occur when aliasing is possible (see Section 3.6).

5.3 Verification Results

A breakdown of the correctness preconditions generated by COVE is summarized in Figure 9. Before explaining the details of Figure 9, we first explain the condition inferred by COVE for Figure 2.

The inferred condition says that the two input vectors v1 and v2 do not alias, that three of the memory dereferences are always zero, and that this example requires unsafe axioms for floating point. Formally, this is expressed as $P \equiv \texttt{restrict(rdi)} \wedge \texttt{restrict(rsi)}$, $\Phi_A \equiv \texttt{8(rdi)} = \texttt{(rsi)} = \texttt{4(rsi)} = \mathbf{0}$, and U; where U says that $\mathbf{0} +_f x = 0$ and $\mathbf{0} \times_f x = 0$ assuming $\mathbf{0}$ is floating-point zero, $+_f$ is floating-point addition, and $\times_f$ is floating point multiplication. The standard calling convention used by gcc/icc ensures that the first argument is in register rdi, the second is in rsi, etc. Using this mapping, a condition can be presented to the user at the source level. The condition P corresponds to $\texttt{restrict(v1)} \wedge \texttt{restrict(v2)}$. However, some conditions contain offsets, e.g., Φ_A requires that memory contains $\mathbf{0}$ at address $\texttt{8(rdi)}$, i.e., $\texttt{v1.z} = \mathbf{0}$. This translation is nontrivial as the binaries do not contain information about types. If the binaries contain debugging information then this mapping is straightforward, but is currently not supported by COVE.

The first column "Benchmark" of Figure 9 is the name of the benchmark. The second column P contains the aliasing conditions inferred for the particular target program. We use α_i to denote the i^{th} argument. The predicate $\rho(\alpha_i, \alpha_j)$ represents adding a restrict annotation to the i^{th} and the j^{th} parameter. A check mark (✓) represents that the inferred aliasing conditions are successfully validated to be true for all possible inputs statically. For example, a 4 byte array with address $\texttt{4(rsi)}$ can never overlap with a dereference to a byte-array at address $\texttt{8(rsi)}$ if the code between the two dereferences does not modify rsi.

The third column Φ_{Align} represents the alignment conditions. A predicate $bX(\alpha_i)$ denotes X byte alignment of the i^{th} parameter. The fourth column, Φ_A represents the conditions that are equalities. The predicate $a(b) = c$ represents that the bit-pattern c is stored at address $a + b$. The abbreviation 0^i represents the hex bit-string $0 \ldots 0$ that has zero repeated i times. An entry $\mathcal{I}$ represents that even though this abstract domain is not present in the condition shown to the user, COVE needs this abstraction to find sufficiently strong invariants. A blank entry signifies that the particular abstraction is neither present in the conditions nor in the invariants. An entry $D_n(\alpha_i)$ is syntax for the condition that the i^{th} parameter points to an $n \times n$ diagonal matrix and hence it abbreviates $O(n^2)$ equations. The odd looking constant $\texttt{0x3ff0}^{23}$ is the bit-representation of 1.0 as an IEEE 754 double precision floating point number.

The fifth column, Φ_B, represents intervals, i.e., inequalities. The predicate $\alpha_i < a$ represents that the 64-bit value in the i^{th} argument is strictly less than a. A check mark in the sixth column, Safe, represents that a safe floating point axiom was required for the proof. This is in contrast to a check mark in the last column that signifies the use of an unsafe floating point axiom. These unsafe axioms used by COVE are valid for real numbers but not for floating point numbers

Benchmark	P	Φ_{Align}	Φ_A	Φ_B	Safe	U
kmeans	$\rho(\alpha_1, \alpha_2)$	$b16(\alpha_1), b16(\alpha_2)$		$\alpha_3 < 2^{24}, \alpha_4 < 2^{24}$		
raysphere	✓		$\alpha_5 = 0^{16}, \alpha_8 = \texttt{0x3ff0}^{23}$ $\alpha_1(\alpha_2) = \alpha_1(\alpha_3) = \alpha_1(\alpha_4)$	$\alpha_1 < 2^{28}$	✓	✓
ex3a	$\rho(\alpha_1, \alpha_2)$	$b16(\alpha_1)$	$\mathcal{I}$			
ex3b	✓	$b16(\alpha_1)$	$\mathcal{I}$		✓	✓
ex3c	✓	$b16(\alpha_1)$	$\mathcal{I}$	$\alpha_2 < 2^{20}$		
test1	$\rho(\alpha_1, \alpha_2)$	$b16(\alpha_1), b16(\alpha_2)$	$\mathcal{I}$	$\mathcal{I}$		
test6	$\rho(\alpha_1, \alpha_2)$	$b16(\alpha_1), b16(\alpha_2)$	$\mathcal{I}$	$\mathcal{I}$	✓	
test10	$\rho(\alpha_1, \alpha_2)$	$b16(\alpha_1), b16(\alpha_2)$	$\mathcal{I}$	$\mathcal{I}$		
test13	$\rho(\alpha_1, \alpha_2)$	$b16(\alpha_1), b8(\alpha_2)$	$\mathcal{I}$	$\mathcal{I}$		
test17	$\rho(\alpha_1, \alpha_2, \alpha_3)$	$b16(\alpha_1), b16(\alpha_2), b16(\alpha_3)$	$\mathcal{I}$	$\mathcal{I}$		
add	✓				✓	
smul	✓				✓	
dotprod	✓				✓	
delta	$\rho(\alpha_1, \alpha_2)$		$8(\alpha_1) = (\alpha_2) = 4(\alpha_2) = \texttt{0}^8$		✓	✓
diag4	$\rho(\alpha_1, \alpha_2, \alpha_3)$		$D_4(\alpha_1), D_4(\alpha_2), D_4(\alpha_3)$			
diag6	$\rho(\alpha_1, \alpha_2, \alpha_3)$		$D_6(\alpha_1), D_6(\alpha_2), D_6(\alpha_3)$			
diag8	$\rho(\alpha_1, \alpha_2, \alpha_3)$		$D_8(\alpha_1), D_8(\alpha_2), D_8(\alpha_3)$			
moss	✓		$\alpha_3 = \alpha_4 = \texttt{0x0}^{14}\texttt{20}$	$\alpha_2 < 2^{16}$		

Figure 9. Conditions required to prove the conditional correctness of STOKE rewrites: aliasing relationships (P), alignment restrictions (Φ_{Align}), equality relationships (Φ_A), inequality relationships (Φ_B), safe floating-point axioms (Safe), and unsafe floating-point axioms (U).

and include associativity of multiplication and addition, distributivity of multiplication over addition, zero as additive identity, and one as multiplicative identity. For benchmarks that do not use unsafe floating-point axioms, COVE's treatment of the floating-point instruction set is sound.

5.4 Discussion

We now give a brief overview of the benchmarks. First, we have the benchmarks from [41], where the authors implement standard compute tasks in C and R and compare performance. We optimize the binaries generated by compiling these C sources with `gcc`. The benchmark `kmeans` [41] is the kernel of an unsupervised learning algorithm for clustering feature vectors. The primary source of optimization for this code is the use of consistently small values in the input data set. A similar property holds for the `raysphere` [41] benchmark, which computes the distance to the closest intersection of a line and a set of spheres. In both cases, STOKE is able to remove many of the computations emitted by `gcc` that are redundant given these ranges. In addition, for the input data sets of `raysphere` from [41], some values are deterministically fixed to either zero or one. This property is readily apparent in the test cases that we provide our binary optimizer and leads to further performance improvements.

The speedups for the other benchmarks of [41] are less than 10% and are omitted. Many of these benchmarks employ indirect addressing and are memory bound. Improving performance for these would require very different algorithms or changing the data structure layout, neither of which are within our present scope. The former would require invariants beyond the current capabilities of COVE(for instance, quantified invariants) and the latter would require global changes to the application instead of local changes to a kernel. Other failure cases of STOKE include simple kernels for which existing compilers produce good code and there are no conditions to leverage.

The benchmarks `ex3a` through `test17` are stand alone kernels. These include array initialization (`ex3a`), reduction over an array (`ex3b`), use of multiple induction variables (`ex3c`), sum of two vectors (`test1`), nested loops (`test6`), conditional vector sum (`test10`), complex indexing (`test13`), and use of several arrays (`test17`). The primary source of speedup on these benchmarks is the clever use of vector instructions, which is safe only under aliasing and alignment restrictions on the inputs.

The next four benchmarks are from a ray tracer [36] that spends the majority of its execution time in the following four operations: vector-vector addition (`add`), scalar-vector multiplication (`smul`), vector dot product (`dotprod`), and the addition of random perturbations to a vector (`delta`). The speedups are due to the use of vector instructions and constant folding. For the dot product, `gcc -O3` produces what appears to be optimal code and as a result none of the approaches described above are able to improve on the result. We plug in these optimized kernel and the result is an end-to-end speedup of over $2\times$ for the whole ray tracer application as a whole.

The `diag4`, `diag6` and `diag8` benchmarks multiply two integer diagonal matrices and save the results in a third. The original code uses the standard $O(n^3)$ matrix multiplication algorithm, and STOKE automatically rewrites the code to use the asymptotically better $O(n)$ algorithm that ignores all the zero entires in the matrix. For these examples, we unroll the triply-nested loop and apply COVE as if the code were straight-line. The only difference between the benchmarks is the dimension of the matrix (4, 6, or 8). Figure 1 shows that `CHECK` is slower than `gcc -O3` for dimension 4, but faster with dimensions 6 or 8. This is because the `CHECK` time is dominated by the checking routine which runs in $O(n^2)$, and for very small inputs this is expensive – but for larger inputs it's much cheaper than running the original code. If the checks are omitted then the speedups are $6\times$, $13\times$, and $20\times$ for dimensions 4, 6 and 8 respectively (`cSTOKE` in Figure 1).

The final benchmark `moss` is a rolling hash function which is the performance bottleneck for MOSS [37], a widely used system for detecting software plagiarism. Such hash functions are also widely used in computational biology and in commercial products that analyze network traffic. STOKE is able to take advantage of the fact that the input array of lexical tokens contains small integers that fit in 16 bits and employ bit-fiddling tricks, that are unsound in general but are correct for the system-wide parameter settings mined from the test cases, to obtain a 35% speedup. In a usual run, MOSS spends about 40% of its execution time in the hash function. The optimized hash function results in a 14% overall speedup in MOSS, a real application.

Finally, we note that in our initial work on the `test10` benchmark (Figure 1), STOKE was able to obtain a nearly $5\times$ speedup over `gcc` and `icc` due to poor branch coverage in the manually written test cases. The correctness preconditions produced by COVE alerted us to this fact and we were subsequently able to add additional test cases to cover the missing paths. We believe that since the conditions are compact (Table 9), developers familiar with the codebase should be able to comprehend the conditions and add the requisite tests. Furthermore, this example reinforces the observation that conditionally correct optimizations can lead to undesirable results if they are generated using test cases that do not accurately represent actual runtime inputs.

5.5 Dynamic Condition Checking

To check preconditions at runtime, we add instrumentation to the `cSTOKE` binaries that checks the inferred condition and runs the conditionally correct code only if the condition is satisfied. It is straightforward to mechanically translate the conditions that we generate to executable code. For our benchmarks, we manually write a straightforward C program that checks the conditions and compile it to generate assembly. We then combine this assembly with the STOKE generated binary and measure its performance (`CHECK` in Figure 1). The advantage here is that this code is correct for all inputs and places no verification burden on the user. Figure 1 shows that for most (but not all) benchmarks the performance degradation is negligible and this correct code is generally significantly better than the correct code generated by `gcc -O3`.

6. Related Work

The literature on equivalence checking is rich and we limit the discussion to formal approaches for x86. Sound equivalence checkers for 64-bit x86 such as DDEC [39] can take hours to validate small programs with less than ten lines of code. Due to the large number of constraints caused by considering all possible program contexts, particularly with respect to aliasing, these techniques do not scale to our benchmarks. Moreover, DDEC works over a fixed abstract domain (affine equalities) and COVE can find invariants over arbitrary abstract domains. This generalization is important because equalities alone are not always sufficient for verification (Figure 9). In contrast, scalable (partial) equivalence checkers for x86 such as [17] make unsound assumptions such as loops do not run for more than two iterations, pointers do not alias, state elements are representable as integers rather than bit-vectors, and that x86 is word rather than byte addressable. Rather than make ad hoc unsound assumptions with no underlying justification, the conditions generated by COVE hold for all test cases. As a result, we have some reason to trust the correctness of the conditions and we believe that sound conditional equivalence is a feasible alternative to unsound approaches.

Conditional equivalence has been suggested in [24] to be a practical alternative to equivalence checking for two versions of a program. The technique presented in [24] is theoretically limited to checking conditional partial equivalence, is purely static, and has been implemented only for loop-free programs. Some verification engines are capable of emitting preconditions under which a proof would succeed if they are unable to verify the correctness of a program [7, 9]. In our work, rather than having verification as the final goal, we use COVE to generate better code.

Systems that trade precision of floating-point programs for performance are orthogonal (we prove bitwise equivalence) and offer no formal guarantees [1, 29, 34, 36, 40].

The generation of invariants from test data was pioneered by Daikon [11], which validates those invariants statically using the unsound static analysis of ESC/Java [31]. Abstract acceleration [21] can be used to obtain invariants for linear loops without fixpoint iterations, although this technique is inapplicable to bit-vectors. YOGI [6] is a weakest precondition based verification engine for Windows device drivers. It specializes the weakest precondition to only include aliasing relationships that are possible in an abstract trace to avoid an exponential increase in constraints due to aliasing. Both COVE and [44] use an iterative process in which candidate invariants that are obtained by abstracting tests are updated using abstract interpretation machinery and decision procedures. The latter uses predicate abstraction to prove safety properties and we use bit-vector abstract domains to prove conditional equivalence.

The VCs that are generated by COVE fall into the category of Horn clauses, for which several solution procedures are available [14, 18]. However, these techniques all rely on theory specific reasoning for unbounded arithmetic (e.g., [18] uses Farkas' lemma), whereas the VCs produced by COVE are over bit-vectors.

Programmer feedback is used to facilitate optimizations in several systems for interactive parallelization: [16, 25] report dependencies that prevent loop parallelization. The system of [43] asks a Java developer to promise restrictions on the side-effects a statement can have. If the programmer agrees, then the system can perform standard optimizations such as dead code elimination. As in our work, these tools can be used to generate conditionally correct code. However, because COVE works in a very different domain (x86 assembly), the optimization opportunities are different. These tools infer assumptions only for a fixed set of transformations and provide no formal guarantees beyond the optimized program working correctly on the given tests. COVE solves a different, and harder, problem by formally proving the conditional correctness of a program of unknown provenance (also see Section 2).

There are other superoptimizers besides STOKE and we survey them here. The first superoptimizer by Massalin [27] did not perform any formal verification and produced sequences of straight line assembly by exhaustive enumeration. Bansal and Aiken [3] implemented an exhaustive enumeration based superoptimizer for x86 and verified the results are correct for all possible inputs using a SAT solver. We believe that COVE can help by providing formal correctness guarantees to Massalin's superoptimizer and allowing Bansal and Aiken's superoptimizer to accept rewrites that their conservative validator rejects. However, since these superoptimizers operate on loop free sequences, the verification task is much easier. Denali [22] and equality saturation [42] use a given set of axioms and enumerate rewrites that are correct for all inputs according to the axioms. These do not benefit directly from COVE as they produce only correct rewrites.

7. Conclusion

Traditional compiler optimizers and superoptimizers are limited in the optimizations they can perform by their lack of knowledge of the actual inputs that may arise in the context of a larger program. We have presented a verification method for automatically constructing such preconditions from observed runtime inputs to a kernel and then proving that the kernel superoptimized to take advantage of those preconditions is equivalent to the original whenever the precondition holds. We show that a binary optimizer based on this verifier can produce optimized kernels that are often multiple times faster than those generated by production compilers.

References

[1] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209, 2010.

[2] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[3] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, pages 394–403, 2006.

[4] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87, 2005.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[6] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. Tetali, and A. V. Thakur. Proofs from tests. *IEEE Trans. Software Eng.*, 36(4):495–508, 2010.

[7] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, pages 132–146, 2012.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming (ISOP 76)*, pages 106–130, 1976.

[9] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, pages 181–192, 2012.

[10] M. Elder, J. Lim, T. Sharma, T. Andersen, and T. W. Reps. Abstract domains of affine relations. In *SAS*, pages 198–215, 2011.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3), 2007.

[12] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings*

of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society, 1967.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.

[14] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[15] B. Hackett and A. Aiken. How is aliasing used in systems software? In *SIGSOFT FSE*, pages 69–80, 2006.

[16] M. W. Hall, T. J. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. H. Paleczny, and G. Roth. Experiences using the parascope editor: an interactive parallel programming tool. In *PPoPP*, pages 33–43, 1993.

[17] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *ESEC/SIGSOFT FSE*, pages 191–201, 2013.

[18] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.

[19] K. Hoffman and R. Kunze. *Linear Algebra*. Prentice Hall, second edition, 1971.

[20] J. A. Howell. Spans in the module (z,n)*. In *Linear and Multilinear Algebra 19*, 1986.

[21] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. In *POPL*, pages 529–540, 2014.

[22] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, pages 304–314, 2002.

[23] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6, 1976.

[24] M. Kawaguchi, S. K. Lahiri, and H. Reblo. Conditional equivalence. Technical report, MSR, 2010.

[25] S. Liao, A. Diwan, R. P. Bosch Jr., A. M. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *PPoPP*, pages 37–48, 1999.

[26] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[27] H. Massalin. Superoptimizer - A look at the smallest program. In *ASPLOS*, pages 122–126, 1987.

[28] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft v0.99.6, 2013.

[29] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, pages 316–333, 2011.

[30] D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, 2004.

[31] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electr. Notes Theor. Comput. Sci.*, 55(2):255–276, 2001.

[32] J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *LCTES*, pages 34–43, 2006.

[33] T. W. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.

[34] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: tuning assistant for floating-point precision. In *SC*, page 27, 2013.

[35] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[36] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs using tunable precision. In *PLDI*, 2014.

[37] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD*, pages 76–85, 2003.

[38] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, 2013.

[39] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven equivalence checking. In *OOPSLA*, pages 391–406, 2013.

[40] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy tradeoffs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[41] J. Talbot, Z. DeVito, and P. Hanrahan. Riposte: a trace-driven compiler and parallel VM for vector code in R. In *PACT*, 2012.

[42] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL*, pages 264–276, 2009.

[43] D. von Dincklage and A. Diwan. Optimizing programs with intended semantics. In *OOPSLA*, pages 409–424, 2009.

[44] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA*, pages 145–156, 2006.

A. Verification Conditions of Figure 3

We explain the verification conditions (VCs) shown in Figure 3. The first VC *Init* states the following:

$$\forall x, x_1, x_2, x_1'.\Phi(x) \wedge x = x_1 = x_2 \wedge x_1' = 0 \Rightarrow I(x_1', x_2)$$

Recall that the states of T and R consist of a value x of a single variable `x`. This constraint is read as follows: we consider an arbitrary state x s.t. the state x satisfies the precondition Φ. The initial value of the variable `x` of the program T is denoted by x_1. The initial value of the variable `x` of the program R is denoted by x_2. Before T and R start executing, both x_1 and x_2 are equal and have the value x. Next, T executes `x=0` and the new value of `x` in T denoted by x_1' becomes 0. The execution of T has reached the loop head. The execution of R reaches the loop head without

changing x. Therefore, when R reaches the loop head for the first time, x has the value x_2. When executions of T and R reach the loop head then their states should satisfy the invariant I. Therefore, under these conditions $I(x'_1, x_2)$ should be true.

The second VC *Ind* states the following (all variables are implicitly universally quantified):

$$I(x_1, x_2) \wedge x_1 < 10 \wedge x'_1 = x_1 + 1 \wedge x_2 \neq 10 \wedge x'_2 = x_2 + 1 \Rightarrow I(x'_1, x'_2)$$

Suppose we start the execution of the loop of T in a state x_1 and the loop of R in state x_2. We do not know anything about x_1 and x_2 except that they satisfy the invariant I, i.e., $I(x_1, x_2)$ is true. Now assume T enters the loop. This can happen only if $x_1 < 10$. Also assume that R also enters the loop. This can happen only if $x_2 \neq 10$. The loop body of T updates the value of x. The new state x'_1 is given by $x_1 + 1$. Similarly, the loop body of R updates the state of x to $x_2 + 1$, denoted by x'_2. Since I is an invariant, these new states should also satisfy I, i.e., $I(x'_1, x'_2)$ should hold under these conditions.

The third VC *Partial* states the following:

$$\forall x_1, x_2 . I(x_1, x_2,) \wedge x_1 \geq 10 \ \wedge x_2 = 10 \Rightarrow x_1 = x_2$$

Suppose we start the execution of the loop of T in a state x_1 and the loop of R in state x_2 such that these states satisfy the invariant I, i.e., $I(x_1, x_2)$ holds. Suppose T exits the loop. This happens only if $x_1 \geq 10$. Similarly suppose R exits the loop. This happens only if $x_2 = 10$. Then both programs T and R terminate. We require that when both programs terminate then they should terminate in equal states, i.e., $x_1 = x_2$ should hold.

The fourth VC *Total* states the following:

$$\forall x_1, x_2 . I(x_1, x_2) \Rightarrow (x_1 < 10 \Leftrightarrow x_2 \neq 10)$$

Suppose we start the execution of the loop of T in a state x_1 and the loop of R in state x_2 such that these states satisfy the invariant I, i.e., $I(x_1, x_2)$ holds. We want to ensure that T exits the loop if and only if R exits the loop, i.e., T and R *mutually terminate*. Since T enters the loop iff $x_1 < 10$ and R enters the loop iff $x_2 \neq 10$, we should have $x_1 < 10 \Leftrightarrow x_2 \neq 10$ for the states satisfying the invariant I.