# From Verification to Optimizations

Rigel Gjomemo[a], Kedar S. Namjoshi[b], Phu H. Phung[c,a], Venkat Venkatakrishnan[a], and Lenore D. Zuck[a]

[a] University of Illinois at Chicago, {rgjomemo,phu,venkat,lenore}@cs.uic.edu
[b] Bell Laboratories, Alcatel-Lucent, kedar@research.bell-labs.com
[c] University of Gothenburg, Sweden

**Abstract.** Compilers perform a static analysis of a program prior to optimization. The precision of this analysis is limited, however, by strict time budgets for compilation. We explore an alternative, new approach, which links *external* sound static analysis tools into compilers. One of the key problems to be solved is that of propagating the source-level information gathered by a static analyzer deeper into the optimization pipeline. We propose a method to achieve this, and demonstrate its feasibility through an implementation using the LLVM compiler infrastructure. We show how assertions obtained from the Frama-C source code analysis platform are propagated through LLVM and are then used to substantially improve the effectiveness of several optimizations.

## 1 Introduction

An optimizing compiler is commonly structured as a sequence of passes. The input of each pass is a *source* code that is first analyzed and, using the analysis information, transformed to a *target* code, which then becomes the source of the next pass in the sequence. Each pass uses static analysis to guide optimization, but the precision of this analysis is limited due to strict time budgets for compiling (e.g., the GCC wiki has as rule 1: "Do not add algorithms with quadratic or worse behavior, ever.") As a result, end users of compilers such as LLVM do not benefit from advances in algorithms for program analysis and verification. These advanced methods are, however, implemented in static analysis tools, which are now widely used to detect programming errors during software development. Examples such tools for C programs include BLAST [10], Frama-C [5], and F-Soft [11], all of which employ SMT solvers to produce high-quality and precise (inductive) invariants.

Static analysis tools are less time-constrained and are thus able to carry out much deeper analysis of program behavior. In this work we explore how the information gathered by such tools can be used to augment the internal analysis of a compiler, and whether this offers any practical benefit. While the compile-time cost of employing additional tools may be high, it is often the case that runtime improvements in optimization outweigh this additional cost, for example, for large frequently used code such as kernels and name servers. One approach is to implement these as optional features inside the compiler.

Yet another option, employed here, is that of importing the analysis results computed by *external* static analysis and software verification tools. There is much to be gained from this modular approach, which decouples analysis from transformation. However, there are two key challenges to be overcome: Linking the output of an analysis tool to the C program representation in the compiler front-end, and propagating the assertions through the program transformations performed at the back-end.

Let us consider the problem of propagating information through a series of optimization passes. The static analysis tool produces information about a source program, say $S$. However, the various passes of a compiler transform $S$ successively into programs $S = S_0, S_1, S_2, \ldots, S_f = T$, where $T$ denotes the final target code. To use information gathered for $S_0$ at the $k^{th}$ compilation stage ($k > 0$), one must have a way of transforming this information into a form that is meaningful for program $S_{k-1}$.

A simple example can illustrate this problem. Suppose that the program $S$ has variables $x$ and $y$, and the static analysis tool concludes that $(x < y)$ is invariant. Now suppose that the first stage of compilation renames $x$ and $y$ to “temporary" variables $t_1$ and $t_2$ respectively. The assertion $(x < y)$ is meaningless for the second compilation stage (from $S_1$ to $S_2$); to be useful, it must be transformed to $(t_1 < t_2)$.

How can assertions be transformed? It is desirable to avoid manually tailoring the propagation of assertions to each transformation, a laborious and possibly error-prone task. Our approach offers a *uniform* method for assertion propagation, which is based on the notion of refinement “witnesses" [14]. Note that when the refinement relation induced by a transformation is available, it can be used to transform any invariant on the source program to an invariant on the target program [1]. We obtain the refinement relation by instrumenting the optimization to produce a refinement relation as it transforms a program. (The validity of the generated relation can be checked independently, using SMT solvers. A valid relation is a “witness" to the correctness of the optimization, hence the name.)

Many standard optimizations only introduce, remove, or rename variables. Thus, witness relations are often conjunctions of equalities between a pair of corresponding source/target variables at a program point (or of the form $v_t = E(V_s)$ where $v_t$ is a target variable, $V_s$ are a source variables, and $E$ is a simple arithmetic expression.) For example, the witness for a dead-variables elimination transformation states that the values of live variables are preserved. In the common case that the invariant depends on a single variable, its propagation can be carried out by simply keeping track of the action that is applied to the variable, without requiring logical manipulations.

In the implementation described in this paper, we handle this common case. The invariants are obtained from the value-range analysis of the Frama-C source code analysis platform [5, 3]. Among other information, Frama-C (via its Value Analysis plug-in) produces invariants which express constant limits on the range

[1] Precisely, if $\varphi$ is invariant for program $S$, and $T$ refines $S$ through relation $W$, then $\langle W \rangle \varphi$ is invariant for $T$.

of a program variable (e.g., $10 \leq x \leq 20$). Such invariants are propagated through LLVM optimizations using a mechanism we describe in Sec. 3. The propagated invariants are used to augment LLVM's own analysis information for optimizations such as instruction combination and buffer overrun checking[2]. Sec. 5 describes some experimental results showing gains vary depending on the relative accuracy of LLVM vs. Frama-C for each benchmark.

The prototype of our implementation is available at `http://www.cs.uic.edu/~phu/projects/aruna/index.html`.

## 2 Approach By Example

We use LLVM [12] as our target compiler infrastructure due to its widespread use in academic and industrial settings, as well as its ability to handle a wide variety of source languages. Among several tools (e.g., [5, 2, 18, 9, 1, 13]) that can be used to obtain external assertions to feed into LLVM, we focus our discussion on Frama-C. In particular, we focus on the use of Frama-C to perform *value analysis*, an abstract-interpretation-based analysis, to obtain various domains of integers (sets of values, intervals, periodic intervals), floating points, and addresses for pointers. The value range analysis results obtained from Frama-C are more powerful than those available in most compilers, and, as we demonstrate, in LLVM.

Consider the code in Fig. 1(a). Even when compiled using the most aggressive optimization scheduler (`-O3` option of Clang), LLVM's optimizer does not detect that the `else` branch in location `L6` is dead (and leaves the branch `L6-L7` intact.)

In Fig. 1(b) we show the ACSL ([6], see also `http://frama-c.com/acsl.html`) assertions produced by the Frama-C's Value Analysis as comments. We note that here examples are given at the C-level for readability, rather then the SSA LLVM bitcode. The assumption of SSA form allows to consider each assertion in a basic block (single-entry single-exit straight line code) to be implicitly the conjunction of assertions preceding it.

We thus omit describing how the assertions produced by Frama-C (comments in Fig. 1(b)) are propagated from the Clang input. The first pass that LLVM performs that is relevant to us is to replace weak inequalities by strict inequalities, possibly at the "cost" of introducing new variables, and replacing, signed comparisons between integers with unsigned ones (subscripted by $u$) whenever the integers are known to be non-negative.

Consider the uncommented lines in Fig. 1(c). There, a new line (L0) is added in which a temporary variable `tmp1` is assigned the value $i - 1$, which, when $i \geq 1$, is non-negative, and hence line L1 does not test it is greater than 0. This allows LLVM to replace the conjunction of the test in L1 by a single unsigned test for `tmp1` $<_u$ 10. Following the quest to replace tests of weak inequality to tests for strict inequalities, LLVM replaces that tests in L2 and L3 by their strict equivalents. Finally, the $j + i$ expression that appears in line L4 (Fig. 1(b)) is

[2] The latter inserts checks; the invariants help identify some which as unnecessary.

replaced by two lines, one (L3.1) that assigns a new `tmp2` the value $j + i$, and the other (L4) tests whether $\texttt{tmp2} \geq k$ (this inequality is left in its weak form, since neither of the operands is a constant.)

Since the original program does not have the new temporaries, there is no value-range analysis for them. However, we can propagate the assertion $i \geq 1 \ \wedge\ i \leq 10$ to the assertion $\texttt{tmp1} = i - 1 \wedge \texttt{tmp1} \geq 0 \wedge \texttt{tmp1} \leq 9$, as appears in L1' of Fig. 1(c). Similarly, using the assertion for $i$ and the assertion $j \geq 5$ (from L2'), we can propagate the assertion $\texttt{tmp2} = i + j \wedge \texttt{tmp2} \geq 6$, which is shown in line L3.1' of Fig. 1(c). Since $k \leq 4$ and $\texttt{tmp2} \geq 6$, the test in L4 can be flagged as trivially true, so that the `else` branch can be eliminated, resulting in the code in Fig. 1(d). (The LLVM passes that accomplish this optimization are instruction combination followed by constant folding followed by jump threading and dead code elimination.)

```
L1: if(i>=1 && i<=10)

L2:   if(j>=5)

L3:      if(k <= 4)

L4:         if(j+i >= k)
L5:            j++;
L6:         else
L7:            j-;
L8: return j;
```

(a) source

```
L1: if(i>=1 && i<=10)
L1':  /*@assert i >= 1 && i<=10*/
L2:   if(j>=5 )
L2':    /*@assert j >= 5  */
L3:     if(k <= 4)
L3':      /*@assert k <= 4*/
L4:       if(j+i >= k)
L5:         j++;
L6:       else
L7:         j-;
L8: return j;
```

(b) with value analysis

```
L0: tmp1 = i-1
L1: if(tmp1 <u 10)
L1':/*@assert tmp1>=0 && tmp1 <=9*/
L2:   if(j>4)
L2':    /*@assert j >= 5  */
L3:     if(k < 5)
L3':    /*@assert k <= 4*/
L3.1:     tmp2 = j+i
L3.1':    /*@assert tmp2 >= 6*/
L4:       if(tmp2 >= k)
L5:     j++;
L6:       else
L7:         j-;
L8: return j;
```

(c) before instruction combination.

```
L0: tmp1 = i-1
L1: if(tmp1 <u 10)
L1':/*@assert tmp1>=0 && tmp1 <=9*/
L2:   if(j>4)
L2':    /*@assert j > 4*/
L3:     if(k < 5)
L3':    /*@assert k < 5*/

L5:       j++;


L8: return j;
```

(d) after using assertions.

**Fig. 1.** Code example illustrating our approach

## 3 External Invariant Usage in LLVM

In this section, we discuss our approach for propagating and using invariants produced by third party verification tools inside LLVM's code transformation passes. As indicated in the introduction, the general approach is based on constructing a refinement witness for each optimization. We describe the theoretical foundations, practical considerations, and the implementation. [14] has a detailed description of the approach while here we only give an overview of it.

*Refinement Relations.* Consider an optimization $\mathsf{opt}$. The optimization $\mathsf{opt}$ can be viewed as a transformer from the source program $S$ into the target program $T = \mathsf{opt}(S)$. Informally, $\mathsf{opt}$ is correct if every behavior of $T$ is a possible behavior of $S$ – i.e., the transformation does not introduce undefined outcomes (such as a divide-by-zero) or non-termination, which do not already exist in $S$. If $S$ is transition deterministic and $S$ and $T$ have identical initial states, this also implies that every behavior of $S$ has a corresponding one in $T$. This notion can be formalized in several ways, depending on the notion of behavior that is to be preserved. We choose to apply a refinement relation that maps $T$-states into $S$-states. A valid refinement relation for a single procedure must:

– Relate every initial $S$-state into an initial $T$-state;
– Relate every initial $T$-state into an initial $S$-state;
– Be a simulation relation from $T$ to $S$. The simulation condition may be single-step simulation or the more relaxed stuttering simulation, and
– Relate every final state $T$-state into a final $S$-state with the same return value(s).

(Note that here we are assuming that both $S$ and $T$ have the same observables and that the return values are observables. Extending the definition for the case where the observables are not the same requires adding a mapping between observables.)

These conditions ensure (by induction) that for any terminating $T$-computation there is a corresponding terminating $S$-computation with same return value, and that every non-terminating $T$-computation has a corresponding non-terminating $S$-computation. With the assumption of transition determinism, this also implies that every terminating $S$-computation has a corresponding terminating $T$-computation.

*Invariant Propagation* Constructing a refinement relation from $T$ to $S$ ensures the correctness of the transformation $T = \mathsf{opt}(S)$. We call such a relation a *witness.* A witness also provides a means to propagate invariants from $S$ to $T$ through the following theorem.

**Theorem 1** *Given a witness $W$ for $T = \mathsf{opt}(S)$, and let $V_S$ (resp. $V_T$) denote $S$'s (resp. $T$'s) variables. Let $\langle W\rangle(\varphi) = (\exists V_S : W(V_T, V_S) \wedge \varphi(V_S))$ (thus, $\langle W\rangle(\varphi)$ is the pre-image of $\varphi$ under $W$). Then for any invariant $\varphi$ of $S$, $\langle W\rangle(\varphi)$ is an invariant for $T$. Moreover, if $\varphi$ is inductive, so is $\langle W\rangle(\varphi)$.*

*Proof.* Consider any execution $\sigma$ of $T$. By definition of $W$, there is an execution $\delta$ of $S$ such that every state of $\sigma$ is related by $W$ to a state of $\delta$. As $\varphi$ is an invariant for $S$, every state of $\delta$ satisfies $\varphi$; hence (by definition), every state of $\sigma$ satisfies $\langle W \rangle(\varphi)$. Inductiveness is preserved since the relation $W$ connects a step of $T$ to a (stuttering) step of $S$. □

*Generating witnesses.* The problem of determining whether a program refines another is, in general, undecidable. However, in the cases we study here, it's usually possible to generate a witness relation by augmenting an optimization $\mathsf{opt}$ with a *witness generator* – an auxiliary function, $\mathsf{wgen}$, that computes a candidate witness, $W = \mathsf{wgen}(T, S)$, for a source $S$ and a target $T$. The tuple $(T, W, S)$ can then be passed to a refinement checker, which checks the validity of $W = \mathsf{wgen}(T, S)$ (by checking each refinement condition). Note that generation and propagation are independent steps.

*Effective manipulation of witnesses.* Obviously, to make the above work in practice it is vital that the generation and propagation of witnesses be carried out effectively. This implies that the witness should be expressed in a logic for which checking is decidable, and for which propagation is computable.

We suppose that witnesses are defined on a basic-block level. Thus, for the check, a program transition is execution of the straight-line (non-looping) code in a basic block. This can usually be expressed as a quantifier-free, array theory formula. (The arrays encode memory.)

What makes this feasible in practice is that the witness relations for standard optimizations can also be expressed in quantifier-free, decidable theories. In fact, they are often simply conjunctions of equalities of the form $v_T = E(u_S)$ where $v$ is either a variable name or memory content and $E(u_S)$ is similar or possibly a simple arithmetic expression over source variable names and constants. For instance, a renaming of variable $x$ to $x'$ has witness $x'_T = x_S$, dead code elimination has a witness which asserts the equality $x_T = x_S$ for all live variables $x$, and so forth. (More examples are given in [14].)

Propagation is the computation of $\langle W \rangle(\varphi)$. For witnesses and assertions expressed in a logic which supports quantifier elimination, one can compute a "closed form" solution. If not, one can still use witnesses to answer queries, as follows. To check whether an assertion $q$ is true in $T$ given the propagated invariant for $\varphi$, one must check the validity of $[\langle W \rangle(\varphi) \;\Rightarrow\; q]$. This is equivalent to the validity of $[\varphi(V_S) \wedge W(V_T, V_S) \;\Rightarrow\; q(V_T)]$. Note that, when $\varphi$ is quantifier-free, so is the second formula. Thus, it is not necessary to carry out quantifier elimination in order to use propagated invariants.

For the experimental work described here, the invariants obtained from Frama-C are of the form $\bigwedge_{v \in V} l_v \leq v \leq h_v$ where the $l_v$ and $h_v$ are integer constants. The transformations of of the form $V_T = \mathcal{E}(V_S)$ where $\mathcal{E}$ is a simple arithmetic expression over $V_S$. Using similarly simple arithmetic manipulations one can compute the pre-image of the invariant. E.g., if $2 \leq x \leq 4$ is $\varphi$ and $y = 2x + 1$ is $W$, then the propagated invariant is $5 \leq y \leq 9$.

Formally, for value-range analysis, $\varphi$ is of the form $\bigwedge_{v \in V_S} l_v \leq v \leq h_v$. We then have:

$$\langle W \rangle(\varphi) \ = \ (\exists V_S : V_T = \mathcal{E}(V_S) \ \wedge \ \bigwedge_{v \in V_S} l_v \leq v \leq h_v)$$

which is of the form $\bigwedge_{v \in V_T} l_v \leq v \leq h_v$. To compute the exact bound for each $u \in V_T$ we need only to track the bounds of the $S$-variables that appear in the r-h-s of $u$'s definition (as per $W$) and do the obvious arithmetic manipulations to obtain the bounds for $u$.

## 4 System Description

### 4.1 Background on LLVM

LLVM's back-end comprises a set of passes that operate on a single static assignment intermediate (SSA) language referred to as LLVM IR or bitcode, which is produced by the Clang front end. There are two types of these passes. One set of passes, called the *analysis passes* gather different types of information about the program, such as loop, alias, and variable evolution, but do not perform any code transformations. The other set, called the *transformation passes* in turn use the information gathered by the analysis passes to reason about and optimize the code. Taken together, they implement several algorithms for program analysis and transformation, such as alias analysis, scalar evolution, instruction simplification, etc.

As mentioned in Sec. 1, recent advances in analysis and verification techniques are not usually included in production compilers due to performance requirements and the implementation effort needed. Our approach aims to address this problem by facilitating the use of results from external verification tools inside the compiler. Using the witness mechanism described in the previous section, we propagate assertions (for which we also use the term annotations interchangeably) obtained from tools such as Frama-C through the various back-end passes of LLVM. By this, we decouple the need for updating the compiler frequently as newer or improved program analysis algorithms become available, as our system is designed to obtain assertions from cutting-edge program analysis tools such as Frama-C. We propagate the assertions to the compiler backend, and employ them in program optimization. However, in realizing this approach, there are a number of practical challenges that must be overcome.

### 4.2 Practical Challenges

These challenges stem from language heterogeneities among the source and intermediate language as well as the code transformations along the sequence of passes. We describe each of these challenges in more detail.

**Source-IR Mapping.** The first challenge faced by our approach is that of propagating invariants from the source code through the front-end to the LLVM IR. In fact, due to the LLVM IR's SSA nature, every source variable can be mapped to several SSA versions in the LLVM IR, and consequently invariants about that source variable must also be bound to those SSA versions. In the case of the C language, an additional problem is posed by its scoping rules where same-named local variables can live in different scopes.

For example, Fig. 2(a) containing a snippet of C source code and Fig. 2(b) containing the corresponding LLVM IR code (simplified for space reasons) up to the comparison `i+size<200`. The two variables `i` declared in two different scopes (L3, L8) are both declared in the entry block in the LLVM IR (L15 for the outer scope `i`, and L16 for the inner scope `i`). However, the invariants (L2, L7) are both with respect to the same identifier `i`, and need to be correctly bound to the corresponding IR variables. Furthermore, these variables are used in different basic blocks; Their values are loaded from memory into SSA variables (L20, L21, L28), which are then used in the following instructions.

```
L0: int j = 0;
L1:  int Arr[N];
L2: /*@assert i>=0 &&
     i<=20*/
L3: int i=getArrNo();
L4: /*@assert size>=0 &&
      size<=100*/
L5: int size=getArrSize(i);
L6: while(j<size)
L7:   /*@assert i>=0
        && i<=99*/
L8:   int i = getArrVal();
L9:   if(i+size<200)
L10:     Arr[j] = setVal(i);
L11:  else
L12:    ...
L13:  j++;
```

(a) C source code.

```
L14: entry: //entry basic block
L15:   %i = alloca i32  //allocate outer i
L16:   %i1 = alloca i32 //allocate inner i
L17:   ...
L18:   br BB %5     //jump to basic block 5
L19: BB:5  //basic block 5
L20:   %7 = load %j
L21:   %8 = load %size
L22:   %9 = icmp slt i32 %8, %9 //j<size
L23:   br %9, ifTrue %10, ifFalse %22
                 //conditional jump
L24: BB:10
L25:   %11 = call @getArrVal() //call function
L26:   store %11, %i1  //store result in %i1
L27:   %12 = load %i1
L28:   %13 = load %size
L29:   %14 = add %12, %13 //i+size
L30:   %15 = icmp slt %14, 200 //i+size<200
L31:   br %15, ifTrue %16, ifFalse %19
                     //conditional jump
```

(b) Corresponding LLVM IR

**Fig. 2.** Example illustrating propagation challenges from C code to LLVM bitcode

**Intermediate Operations** . Another problem introduced by LLVM's IR is its three address code nature: Consider the test `i+size < 200` in L9 of Fig. 2(b). It is compiled to two loads (L27, L28) and an addition (L29), followed by the comparison L30, which based on the invariant information will always be true.

In order to fold it and set the value of `%15` to `true`, it is necessary to propagate the value-range information on `size` and `i` (therefore on `%12` and `%13`) to the value-range information on `%14`.

**Code Transformation** . The LLVM IR undergoes transformations along the sequence of passes, and the assertions must be transformed accordingly. Consider, the transformation of `L1` from Fig. 1(b) to Fig. 1(c), where the test $(i \geq 1) \wedge (i \leq 10)$ is replaced by the assignment `tmp1`$= i - 1$ followed by the test `tmp1` $<_u$ 10. This entails computing the bounds on `tmp1` and verifying the correct use of $<_u$. Other passes, such as `mem2reg`, which promotes memory to registers, may make even more drastic changes such as removing `load` and `store` operations or introducing $\phi$ functions.

### 4.3 System Architecture

The architecture of our system is depicted in Figure 3. The input to the system is a C source program and a set of invariants generated for that program by verification tools. The C source code is annotated with the invariants and the annotated source code is compiled by the front end into LLVM IR. Before being passed to the standard LLVM backend, the IR program is processed by two LLVM passes that we wrote: *Annotation Mapping* and *Annotation Propagation*. The former binds the assertions contained in the annotations to the SSA versions of the source code variables, while the latter combines the assertions and propagates them to the intermediate operations that use those variables. These two passes are run before any other pass, in order to operate on the IR program version produced by the front end' s code generation step.

We assume that every optimization pass generates a witness of its correctness (see [14]), and, with the assertions produced by the external static analysis tools, the witnesses are propagated to the passes that can utilize them. The experiments described in this paper use per-variable value range assertions, hence the assertions propagated are conjunctions of equalities (as described in Sec. 3) and are easy to implement, without the need for explicit logical manipulation, as explained in Sec. 3.

The value range invariants are currently used in three optimizations, namely *array bounds check* insertion, *integer overflow check removal*, and *instruction combination*. The first is a set of passes that insert run time checks for every array reference in a program to detect out of bounds accesses. The second is an optimization pass that we wrote to safely remove run time checks inserted by LLVM when it is invoked with the bounds-checking option. The third is a modified *instruction combination* pass that operates on comparison simplifications. We describe each component of our system in more detail next.

*CIL-based rewriter.* Our approach uses a subset of ACSL to express assertions, which are supplied to the framework through an input file (see Figure 1). ACSL allows for a wide variety of first order global and statement assertions. For

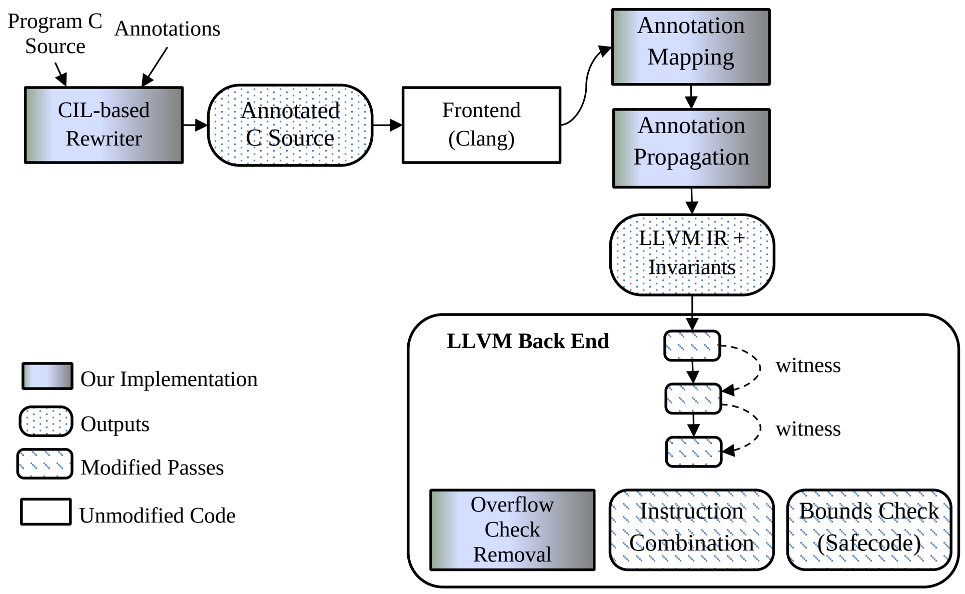

**Fig. 3.** System Architecture

instance, this includes value-range assertions about variables and ghost variables in each program location of the type $a \geq 0 \ \wedge \ a \leq 10$ and $a = 10$.

One of our goals is to support a wide variety of program analysis tools as sources for assertions. A clean, compiler-independent way to do this is by storing the assertions in 'dummy' string variables before the corresponding instructions in the source code. These variables are specially named so that they do not interfere with the existing program variables. As the assertions are encoded as assignments to special variables, these assertions are propagated to the LLVM IR. To do this, we implemented a rewriter based on CIL [16] to inject assertion strings into C source files. The result of this rewriting is shown in Listing 1.2, where statement 1 is the injected one.

**Listing 1.1.** C Source.

```
1 int* a=malloc(X*Y
2        *sizeof(int));
3 *(a+X+Y-1)= Z;
4 m= max(a, X*Y);
```

**Listing 1.2.** Annotated Code.

```
1 char *acsl_b_1="X==2 && Y==4";
2 int* a=malloc(X*Y*sizeof(int));
3 *(a+X+Y-1)= Z;
4 m= max(a, X*Y);
```

*Annotation Mapping.* The goal of the *Annotation Mapping* pass is to bind every invariant written in terms of source variables to the correct SSA variable versions in the IR code. These variables are typically created by LLVM `load` instructions before being used. To achieve its goal, the *Annotation Mapping* pass consults the debugging information, which contains mappings between `load` instructions and source code variables as well as information about the source scope of the original variable. The scope information is used to disambiguate between the SSA versions of the same-named source code variables.

Our pass binds invariants to `load` instructions by attaching to them LLVM metadata containing the upper and lower bound of the range of the corresponding variable. For instance, with respect to Fig. 2(b), the invariant ($\%12 \geq 0 \wedge \%12 \leq 99$) is attached as metadata to the instruction L27. These metadata are valid until the next `store` instruction to the same variable or until a new invariant about the same variable appears. The metadata are currently per-instruction and are not modified by the normal transformation passes, except when the instruction and its uses are removed, in which case the metadata are also lost. In those cases when an instruction or group of instructions are replaced by simpler ones, the invariant information contained in their metadata is combined and added as new metadata to the target instructions. Metadata are orthogonal to the IR and the choice of using them for storing invariants enables us to implement a large range of additional logic and witness propagation while minimizing the interference with the outputs of standard LLVM passes.

*Annotation Propagation.* Starting from the `load` metadata, this pass propagates range information to the other instructions in the IR code, especially those that compute intermediate results. With respect to Fig. 2(b), this pass combines the invariant on L27 ($\%12 \geq 0 \wedge \%12 \leq 99$) and the one on L28 ($\%13 \geq 0 \wedge \%13 \leq 100$) to obtain a new invariant that is attached to L29 ($\%14 \geq 0 \wedge \%14 \leq 199$).

Currently, the supported LLVM instructions include `add, sub, store, mul, sdiv, udiv, sext, zext`, and `getelementptr`.[3] The binary arithmetic instructions are supported via an LLVM class, `ConstantRange`, which is used to represent constant ranges and provides the capability to perform such arithmetic operations on ranges. The `sext` and `zext` operations on ranges yield the same range. For the `getelementptr` operation, which takes in input an array reference and an index and returns a pointer to the corresponding array element, we use two types of metadata, one contains the index range and the other contains the size of the array, if known at compile time. This latter type of metadata is widely used in the bounds check elimination pass.

## 4.4 Optimizations

*Bounds Check Removal (Safecode).* Safecode [7] is a tool composed as a sequence of passes, which insert calls to run time bounds checking procedures before every array access in the LLVM IR. While ensuring safety of memory accesses, however, it introduces substantial overhead at run time. More specifically, these functions are inserted before every `getelementptr` instruction and `store` or `load` instruction that makes use of the value returned by `getelementptr`. For instance, consider Fig. 4, which contains the rest of the LLVM IR code that follows Fig. 2(b) and which contains an array access. In this code, a function

[3] http://llvm.org/docs/LangRef.html

call (L38) is inserted after the `getelementptr` instruction (L37) and before the `store` instruction (L39). In addition to bounds checking, these functions perform several pointer arithmetic operations increasing the program's execution costs.

If, however, at compile time, it can be proved that an array access will never be out of bounds during execution, then the bounds checks on that access can be removed. To do so, in our implementation, we modify the Safecode passes to consult the two types of metadata (related to the index range and to the array size) for the `getelementptr` instructions. If, using this information, it is determined that out of bounds access is not possible, then the calls to the bounds checking functions are removed.

```
L32: BB:16
L33:   %17 = load %i1
L34:   %18 = call @setVal(%17) //setVal(i)
L35:   %19 = load %j
L36:   %20 = sext %19 to i64
L37:   %21 = getelementptr %array, %20 //Arr[j]
L38:   %el = call @checkGEP(%array, %21, 200)
L39:   call @checkStore(%array, %el, 200)
L40:   store i32 %18, i32* %21 //Arr[j] <- setVal(i)
L41:   br BB %19
L42: BB:19
L43:   ...//j++
L44:   br %5
```

**Fig. 4.** An Example (continued)

*Integer Overflow Check Removal.* Another use of the range information is to remove unnecessary integer overflow checks inserted by some of the `-fsanitize` family of Clang options. As shown in Fig. 5, these options transform every operation that may result in overflow into a procedure call (L3), which performs the operation and sets an overflow flag. If overflow occurs, the control flow jumps to an error handling procedure (basic block `handle_overflow`), otherwise execution proceeds normally (basic block `cont`).

The key intuition here is that if range information is available at compile time for the operands, then the possibility of overflow may be checked at compile time and unnecessary checks will be removed. In fact, each check transforms simple (and frequent) operations like additions into procedure calls and comparisons, incurring in high performance costs. Our pass, which at compile time is run after the `-fsanitize` passes, checks the possible value range of the result and removes the integer overflow procedure calls if it determines that overflow is not possible.

*Instruction Combination.* Instruction combination is a powerful transformation pass in LLVM, which simplifies instructions based on algebraic properties. One instruction on which the pass operates is the *integer comparison* instruction

```
L1:   %12 = load %i1
L2:   %13 = load %size
L3:   %14 = call @llvm.sadd.with.overflow(%12, %13)
L4:   %15 = extract overflow flag
L5:   br %15, ifTrue %cont, ifFalse %handle_overflow
L6  handle_overflow:
L7:   call usban_handle_overflow()
L8:   br %cont
L9: cont:
L10   ...
```

**Fig. 5.** Integer Overflow Detection Example

(`icmp`), which performs comparisons between integers. The result of this instruction is placed in a boolean variable, which is usually consulted by branching instructions to issue jumps to the true or false target basic blocks.

The use of range information in this case is fairly straightforward once it is available to the pass. In particular, if the ranges of the two variables being compared at run time are known at compile time and disjoint, then the comparison result is folded to either true or false. With respect to Fig. 2(b), using the range information on the variable `%14`, the comparison is folded and L31 is transformed into (`br TRUE, ifTrue %16, ifFalse %19`). Next, the standard jump-threading pass replaces L31 with an unconditional jump (`br %16`), while the dead code elimination pass removes L30 and L29, which are not used anymore.

## 5 Evaluation

In this section, we present our experimental results on above mentioned optimization passes in LLVM using our framework. We use a set of small to medium size benchmarks that are listed in Table 1.

| Benchmark | Brief description | LoC | Frama-C (ms) |
|---|---|---|---|
| Susan[7] | Low Level Image Processing | 1463 | 528 |
| NEC Matrix[8] | Matrix operations | 113 | 2 |
| CoreMark[9] | CPU performance with list and matrix operations | 1831 | 251 |
| Linpack[10] | Floating point computing power | 579 | 11044 |
| Dijkstra[7] | Network routing | 141 | 6 |
| Mxm[10] | Matrix-matrix multiplication problem A = B * C | 373 | 9 |

**Table 1.** Benchmarks with brief description and size information

*Experimental Methodology* As mentioned earlier, we use the Frama-C tool [5] as our input source for assertions for the benchmarks. Frama-C is based on abstract interpretation and it can be configured with different options that control its running time and accuracy. The running times of Frama-C on the benchmark files, with its default options, are displayed in Table 1. For the Linpack benchmark instead, Frama-C was configured to unroll loops 1000 times. In particular, we extract the value range information from Frama-C's internal state, and its translation to ACSL format. To this end, we have implemented a Frama-C plug-in, which visits the program's AST tree and the value analysis plug-in's state, and writes the value ranges available at each program point in a separate annotation file. Using these assertions, the CIL-based rewriter transforms the C source file by injecting these assertions at the corresponding program locations as described in Section 4.3. After the rewriting step, the annotated sources are passed through the Clang front end of the LLVM compiler.

In our experiments, we report on the optimizations to the benchmarks. Our comparisons are made by running the benchmarks under the unmodified LLVM that does not include our optimizations. We report both the percentage of checks that are removed using our framework (a *static* measure of improvements), and also the percentage savings in running time (a *runtime* measure of improvements). Our runtime tests were performed on a GNU-Linux machine running the Ubuntu distribution 12.04, with Intel Xeon CPU at 2.40GHz.

### 5.1 Array bound-check optimization

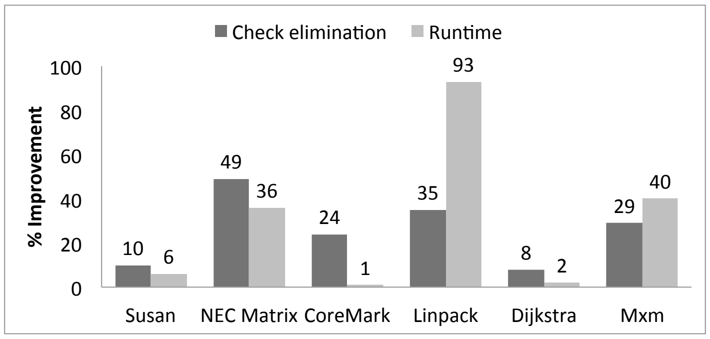

**Fig. 6.** Percentage check elimination and Runtime Improvement of Boundcheck (Safecode) optimization.

7 `http://www.eecs.umich.edu/mibench/source.html`

8 Part of the NEC Lab benchmarks for F-soft [11]

9 `http://www.eembc.org/coremark/download_coremark.php`

10 `http://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html`

Fig. 6 shows our check elimination and runtime improvement results over the benchmarks. Each benchmark is presented in two bars for check elimination and runtime improvement percentage. The check elimination improvements are observed by counting the number of checks on original code and comparing them with the results on optimized code.

As illustrated by Fig. 6, there is a wide variety in our improvement results. This variety is due to several factors. Some of these include the following: (a) Frama-C is not able to produce assertions for every array access as it is not possible to determine the size of those arrays at compile time, or (b) because our prototype does not support certain types of array accesses yet. In particular, we noticed that in some benchmarks, it is not possible for Frama-C to determine the size of the arrays in the case these array initializations depend on some runtime arguments. In these cases, the improvement results are not significant. For example, among the benchmarks, Dijkstra (a network routing algorithm) only obtains 8% of bound check elimination since its computations heavily depend on runtime arguments which are based on the input data. In contrast, with a good quality of assertions, our approach obtains very appealing improvements. For instance, NEC Matrix gains the best improvement of 49% in our experiments. This is due to fact that the benchmark has many array accesses, and most of which have good assertions from Frama-C. In addition, the runtime improvements depend on the location of the eliminated checks. If they are located in portions of code that are not executed very often (e.g., initialization code in CoreMark), then the runtime improvement is not significant. If, however, they are located in a portion of the code that is executed often (e.g., Linpack) the improvements can be significantly better. It is worth noting that our optimizations are done based on Frama-C's sound analysis, and therefore carry the same guarantees of the safety of array accesses under LLVM's Safecode bounds checking.

### 5.2 Integer Overflow Check

The chart in Fig. 7 illustrates the improvement on the integer overflow check elimination by our framework. Similar to the previous experiment, here too we report on both check elimination and runtime improvements for the benchmarks. The checks are inserted for the LLVM IR's operations of multiplication (`mul`), addition (`add`), and subtraction (`sub`). As shown in the figure, the improvement ranges from 7% (Susan) to 60% (Mxm) of checking code of integer overflow on the benchmarks. As before, the improvements are dependent on the quality of assertions and the benchmark itself. For Susan, most of values of variable depend on runtime arguments so that we do not get good assertions from Frama-C. On the other hand, Mxm benchmark contains a large percentage of integer computations and these computation variables have good assertions from Frama-C.

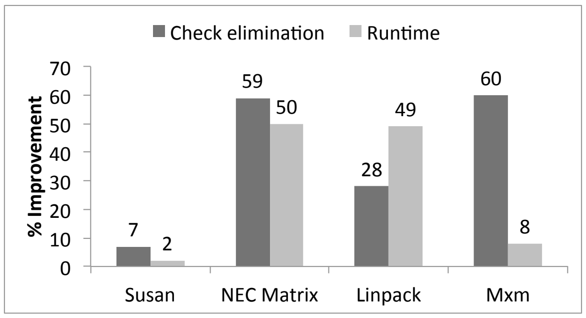

**Fig. 7.** Percentage check elimination and runtime improvement of Integer Overflow Check Optimization.

### 5.3 Instruction Combination

To take advantage of range information for folding comparisons as described in our examples, we have modified the Instruction Combination pass in LLVM (`-instcombine`). We have tested our implementation with a number of small examples and our implementation is able to perform the optimization successfully. In our experiments with the above benchmarks, the opportunities for applying these optimization does not arise. This is due to the fact that, in these benchmarks, Frama-C does not produce assertions for branch expressions that lead the branch condition to be evaluated to either true or false at compile time. We have noticed that larger benchmarks such as `bind` and `gcc` and `oggenc` provide opportunities for such optimization, but Frama-C does not successfully compile these benchmarks out of the box. We have been working with the Frama-C development team to get assertions on these larger benchmarks.

## 6 Related Work

To the best of our knowledge, our work is the first that uses analysis information derived by third party tools, which are not as restricted as production compilers, to improve compiler optimizations. The key issue is that of invariant propagation. Our implementation results show that, for the common case of single-variable invariants, we can carry out this propagation quite simply, which results in substantial improvements to compiler optimizations. Propagation lets a compiler use the results of sophisticated program analyses without incurring the cost of the analysis during compilation. We strongly believe that this is a promising approach that will has much potential for improvements.

There are several tools and compiler extensions which combine sophisticated analysis with code transformation. Examples are Klee [4] (for symbolic execution), Polly [9] (for polyhedral optimization), CCured [15] (for bounds checking) and IOC (Integer Overflow Checker) [8]. The key new element introduced by our work is in loosening the coupling between analysis and optimization, i.e., providing a mechanism for introducing the results of *any* sound program analysis into a standard compiler (or, more generally, a program transformation), without requiring that the analysis be built into the compiler.

The idea of propagating assertions through a witness mechanism was first introduced in [14]. Witness generation is itself a variant of the translation validation framework introduced in [17] and developed by several researchers (cf. the citations in [14]). Just like the translation validation framework, it does not depend on specific passes (even though the generation of witnesses, on which we do not focus here, does depend on specific optimizations), but it depends on the ability to “tweak" the compiler, as well on the assumptions that each optimization is a separate, easy-to-identify, pass.

## 7 Conclusion and Future Work

We describe a methodology, supported by tools, for enabling compilers to use the results of external program analysis tools to enable better optimizations. The assertions produced by the external tools are propagated, through the witness approach, through the LLVM optimizations passes. We demonstrate the methodology by improving three LLVM optimizations using the Frama-C value analysis plugin. We are currently expanding our approach to encompass other static analyses as well as targeting other LLVM passes, such as scalar evolution and loop optimizations.

*Acknowledgements* We thank Drew Dean for pointing us to this topic in the context of the DARPA CSFV project. Thanks are due to Ted Ballou and Anokh Kishore for their help with the experiments. We also thank Jens Palsberg for many useful comments on this work. Phu Phung is supported by the Swedish Research Council (VR) under an International Postdoc grant. This material is based on research sponsored by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

## References

1. A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, (PASTE)*, pages 43–48, 2007.
2. A. Albarghouthi, A. Gurfinkel, Y. Li, S. Chaki, and M. Chechik. UFO: Verification with interpolants and abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *Lecture Notes in Computer Science*, pages 637–640, 2013.
3. R. Bonichon and P. Cuoq. A mergeable interval map. *Studia Informatica Universalis*, 9(1):5–37, 2011.

4. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.
5. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C, a software analysis perspective. In *International Conference on Software Engineering and Formal Methods (FMICS'12)*, 10 2012.
6. M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *28th Annual ACM Symposium on Applied Computing, SAC*, pages 1230–1235, 2013.
7. D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. Technical report, Shanghai, China, May 2006.
8. W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *34th International Conference on Software Engineering*, ICSE '12, pages 760–770. IEEE Press, 2012.
9. T. Grosser, A. Größlinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
11. F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-SOFT: Software Verification Platform. In *17th International Conference on Computer Aided Verification (CAV)*, pages 301–306. Springer-Verlag, 2005.
12. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004. Webpage at llvm.org.
13. K. R. M. Leino. Extended Static Checking: A ten-year perspective. In *Informatics – 10 Years Back. 10 Years Ahead.*, pages 157–175, 2001.
14. K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. In *Proc. $20^{th}$ Static Analysis Symposium*, volume 7935 of *LNCS*, pages 304–323, 2013.
15. G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.
16. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *11th International Conference on Compiler Construction (CC)*, pages 213–228. Springer-Verlag, 2002.
17. A. Pnueli, O. Strichman, and M. Siegel. Translation validation: From DC+ to c*. In *Applied Formal Methods - International Workshop on Current Trends in Applied Formal Method(FM-Trends)*, volume 1641 of *LNCS*, pages 137–150. Springer, 1998.
18. T. Teitelbaum. Codesurfer. *ACM SIGSOFT Software Engineering Notes*, 25(1), 2000.