**POLITECNICO DI MILANO**
**Corso di Laurea Magistrale in Ingegneria Informatica**
**Dipartimento di Elettronica, Informazione e Bioingegneria**


# Improving the Compilation process using Program Annotations

**Relatore: Prof. Giovanni Agosta**
**Correlatore: Prof. Lenore Zuck**

**Tesi di Laurea di:**
**Niko Zarzani, matricola 783452**

**Anno Accademico 2012-2013**

*Alla mia famiglia*
*Alla mia ragazza*
*Ai miei amici*

# Ringraziamenti

Ringrazio in primis i miei relatori, Prof. Giovanni Agosta e Prof. Lenore Zuck, per la loro disponibilità, i loro preziosi consigli e il loro sostegno. Grazie per avermi seguito sia nel corso della tesi che della mia carriera universitaria.
Ringrazio poi tutti coloro con cui ho avuto modo di confrontarmi durante la mia ricerca, Dr. Venkat N. Venkatakrishnan, Dr. Rigel Gjomemo, Dr. Phu H. H. Phung e Giacomo Tagliabure, che mi sono stati accanto sin dall'inizio del mio percorso di tesi.
Voglio ringraziare con tutto il cuore la mia famiglia per il prezioso supporto in questi anni di studi e Camilla per tutto l'amore che mi ha dato anche nei momenti più critici di questo percorso. Non avrei potuto superare questa avventura senza voi al mio fianco.
Ringrazio le mie amiche e i miei amici più cari Ilaria, Carolina, Elisa, Riccardo e Marco per la nostra speciale amicizia a distanza e tutte le risate fatte assieme.
Infine tutti i miei conquilini, dai più ai meno nerd, per i bei momenti passati assieme. Ricorderò per sempre questi ultimi anni come un'esperienza stupenda che avete reso memorabile. Mi mancherete tutti.

# Contents

# List of Figures

# List of Tables

# Riassunto

Gli attuali compilatori evitano analisi che richiedono algoritmi ad alta complessità. Al fine di permettere una veloce compilazione alcune delle ottimizzazioni possibili grazie a queste più costose analisi vengono sacrificate. Per esempio nelle GCC Developer Guidelines è espressamente vietata l'introduzione di algoritmi di complessità quadratica o peggiore in aggiunta ai necessari già presenti. Al contrario i software per l'analisi statica e per la verifica formale del codice sono in grado di generare informazioni di alta precisione al costo di algoritmi di complessità maggiore. In questa tesi descriviamo come poter ottenere un compilatore che sia in grado di migliorare alcuni aspetti della compilazione facendo uso delle informazioni esterne provenienti da strumenti per la verifica formale e l'analisi del codice. Abbiamo focalizzato l'attenzione sull'uso di queste informazioni sia per migliorare le già esistenti ottimizzazioni del codice, sia per ridurre il costo nel garantire che il codice sia esente da vulnerabilità di sicurezza di tipologia buffer overflow. Il nostro progetto *Aruna* è stato sviluppato usando l'infrastruttura LLVM e si propone come target programmi scritti in C. Inserendo annotazioni nel codice sorgente, *Aruna* utilizza gli invarianti forniti da tool esterni permettendone la propagazione successiva sino al back-end del compilatore al fine di migliorare il processo di compilazione. Questo lavoro ha lo scopo di realizzare il pezzo mancante fra gli attuali compilatori e gli strumenti di analisi esterni. Proprio per questo il framework *Aruna* è stato sviluppato in maniera modulare in modo da permetterne un utilizzo futuro con annotazioni provenienti da diversi tipi di tool esterni e per ottimizzazioni diverse in aggiunta a quelle trattate in questo scritto.

# Abstract

Current compilers typically avoid high-complexity analysis algorithms. For the sake of a fast complication they eschew high-precision analysis that may compromise optimizations. For instance, the GCC Developer guidelines prohibits the introduction of even quadratic algorithms into the compiler code. However, current static analyzers and formal verification tools are able to produce high-precision informations using high-complexity algorithms. In this work we focus on how the informations manually provided by the developer, by crowd source formal verification or by static formal verification tools can be used to help the compilation process. The work described in this thesis aims at obtaining a compiler that performs optimizations based on high-precision analysis obtained with tools for static analysis and formal verification. Our framework *Aruna* is a project whose goal is to obtain this for C programs compiled using the LLVM infrastructure. *Aruna* is built on top of the LLVM architecture and can be useful to software developers as it will improve performance while significantly enhancing security. Using source instrumentation, *Aruna* augments a given program with externally supplied assertions. Subsequently, a modified front-end of LLVM propagates these invariants to the optimization back-end. We therefore view our approach as providing a crucial piece that is missing from current production compilers. The framework modularity ensures a future use with external annotations coming from different tools and for different improvements on the back-end side.

# Chapter 1

# Introduction

Many programming tools can help the programmer during the application development. Among them there are tools used to test program correctness, tools for program safety and tools for code optimization and transformation.

During the transformation of the source code into a target language compilers enable the code generated to work more efficient and use fewer resources. In addition they can also be used to enable program safety by inserting run-time checks to avoid common security flaws. Current compilers typically avoid high-complexity analysis algorithms. For the sake of a fast complication they eschew high-precision analysis that may compromise optimizations. For instance, the GCC Developer guidelines prohibits the introduction of even quadratic algorithms into the compiler code[15].

Formal verification tools can be helpful in proving the correctness of software expressed as source code. They allows to verify that the source code complies with a provided formal specification. These tools implement powerful analysis that can compute information automatically from the source code of a program, allowing the programmer to verify that the code satisfies a formal specification. This in turn, enables program verification faster and less risky than code review. Much research was done in program analysis in order to obtain formal proof of program behaviors. Unlike analysis implemented in compilers, this kind of analysis has a high time and space complexity.

Programmers often insert runtime checks in their programs and build assertion enabled version of their projects. This version is tested at runtime and programmers expect the behavior of the program to comply with the properties specified in those assertions before the code will be

removed in production.

In this work we focus on how the informations manually provided by the developer, by crowd source formal verification or by static formal verification tools can be used to help the compilation process. The work described aims at obtaining a compiler that performs optimizations based on high-precision analysis obtained with tools for static analysis and formal verification.

Our framework *Aruna* [1] is a project whose goal is to obtain this for C programs compiled using the LLVM infrastructure. *Aruna* is built on top of the LLVM architecture and can be useful to software developers as it will improve performance while significantly enhancing security. Using source instrumentation, *Aruna* augments a given program with externally supplied assertions. Subsequently, a modified front-end of LLVM propagates these invariants to the optimization back-end. We therefore view our approach as providing a crucial piece that is missing from current production compilers. The modularity of the framework ensures a future use with external annotations coming from different tools and for different improvements on the back-end side.

**Thesis Organization.** The dissertation is organized as follows:
We describe what we mean for annotated code, its major sources and some of the basic compiler concepts needed to understand the design choices of this work in Chapter 2.
The overview of the infrastructure that enables the propagation of assertions and their use is described Chapter 3.
Chapter 4 describes in detail the framework implementation.
With respect to the normal LLVM infrastructure, in Chapter 5 we present an evaluation of this work.
Chapter 6 discusses the limitations and future work, and we conclude in Chapter 7.

[1] Aruna is the Charioteer of the Sun in Hindu Mythology as well as a DC Comics shapeshifter. The name was chosen to reflect the various shapes of annotations that will be processed by the tool.

# Chapter 2

# Background

Annotated code contains useful information that is often ignored by compilers. These information may came from programmers, automatic analyzers or crowd source tools. When these annotations are written in a standard annotation language compilers may be instrumented to take advantage of them. However this information needs to be propagated inside the existing compiler architecture so that the back-end can make use of them.

In this chapter we introduce the basic concepts underlying the rest of this thesis. We will give a general overview of what we mean by *program annotation* and the different sources of annotated code. We present a taxonomy where the sources differs by the way they are generated in Section 2.2. In Sections 2.2.1, 2.2.2 and 2.2.3 we inspect real world sources of annotations. The main task of *Aruna* is the propagation of the information inside the annotations to the back-end optimization passes, where they can be used. This is the reason why in Section 2.4 we briefly cover the necessary concepts of a compiler architecture that motivate the design choices during the implementation of *Aruna* . To conclude the background chapter in Section 2.4.2 we cover the building blocks of Low Level Virtual Machine (LLVM) on top of which which our framework is built.

## 2.1 Annotated Code

Programs may contain useful information that is often ignored by modern compilers. This information may be there as preprocessor directives, for code debugging, for code documentation and organization or may be the result of the analysis of an external tool. In general *program anno-*

*tation* refers to annotation either in Source Code (SC) or Intermediate Representation (IR) with additional informations that does not affect the semantic of the program. Usually these annotations are inserted as comments, pragmas or special functions in the program SC or as metadata information in the program IR.

## 2.2 Sources of annotated code

There are several sources of annotations that can be used to improve compiler optimizations. We choose to classify them by the way they are generated:

- *Manually Generated*: this kind of annotations requires the programmer to specify the information inside the program SC.

  The programmer is responsible for the correctness of the information contained in each specified annotation.

- *Automatically Generated*: this kind of annotations comes from analysis programs and does not require any human effort during the generation of the annotated code. They can be generated both inside the program SC or IR.

  The correctness of the information contained in each annotation relies on the correctness of the program analysis and its implementation.

- *Crowd Generated*: this last kind of generation is somewhat in between manual and automatic generation but does not require any effort from the programmer. It relies on humans generating formal proved annotations by means of software tools. Therefore the program annotation is crowdsourced. These tools may not directly expose the program to the user, for example they may show an equivalent model with a different representation.

  The correctness of the information contained in the annotation relies on the correctness of the tool used by the crowd of human annotators.

Real example of annotations are the ones coming from the programmer itself, Crowd Source Formal Verification (CSFV) and program analyzers based on abstract interpretation and lazy abstraction. In the next subsections we are going to give some examples of program annotation and their corresponding annotated code.

### 2.2.1 Programmer's annotation

Developer's annotation are usually inserted in the code in order to support the software design and to test the correct program behavior. Common annotations that can be used to improve optimizations are the ones coming from the design by contract paradigm, debug functions and preprocessor directives. We will see them in detail in the following subsections.

#### Design by contract

Software designers commonly define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. In languages such as Eiffel[16] it is part of the design process and required in the implementation.

Listing 2.1 presents a trivial C example showing a precondition that can be used to remove an if-else statement.

*Listing 2.1: Precondition useful for optimization*

```
1   /*@
2           requires x > y;
3           requires y > 0;
4   */
5   int foo(int x, int y){
6           if ( x > 0 ){
7                   return x+y;
8           }
9           else {
10                  return -1;
11          }
12  }
```

#### Assertions

An assertion is a predicate (a statement containing a boolean expression) placed in a program to indicate that the developer believes the predicate to hold when control reaches this location. An assertion that evaluates to false at run-time typically causes execution to abort. Usually, developers expect the behavior of the program to comply with the properties specified in those assertions, so that code can be removed in production. The correct execution of the program relies on these properties, therefore they may be used to improve code optimizations.

In Listing 2.2 we present a trivial C example showing an assertion that can be used to remove an if-else statement.

*Listing 2.2: Assertion useful for optimization*

```
1       int foo(int x, int y){
2               assert( (x > y) && (y > 0) );
3               if ( x > 0 ){
4                       return x+y;
5               }
6               else {
7                       return -1;
8               }
9       }
```

### Pragmas

The *#pragma* directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. The basic form of a pragma is showed in Listing 2.3.

*Listing 2.3: Basic Pragma Form*

```
1       #pragma token-string
```

The *#pragma* directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C, Objective-C and C++ languages. For instance GCC has some function-specific option pragmas[22] and loop-specific pragmas[23]. An interesting set of pragmas that are currently supported by LLVM are the Open Multi-Processing (OpenMP) pragmas. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. The example in Listing 2.4 shows a section of code that is meant to run in parallel and is marked accordingly with a preprocessor directive that will cause the threads to form before the section is executed.

*Listing 2.4: OpenMP Pragma Example*

```
1       ...
2       const int N = 1000;
3       int i, array[N];
4
5       #pragma omp parallel for
6       for (i = 0; i < N; i++)
7           array[i] = i * 3;
8       ...
```

Currently the definition of new pragmas inside LLVM is cumbersome as mentioned in *An experimental framework for Pragma handling in*

*Clang*[24]. This is why in our implementation we will use an alternative way to propagate the annotations without the use of pragmas.

### 2.2.2 Crowd Source Formal Verification

Nowadays crowd-sourcing problems that are hard to analyze seems to be a promising idea. Many interesting applications[1] such as OpenStreetMap and Recaptcha rely on volunteer work to solve complex analysis.

An example of crowd annotation generation is CSFV[9], a program that seeks to make formal program verification more cost-effective by reducing the skill set required for verification. An automated game-level builder transforms the program verification models into compelling games. The CSFV annotation process is shown in Figure 2.1.

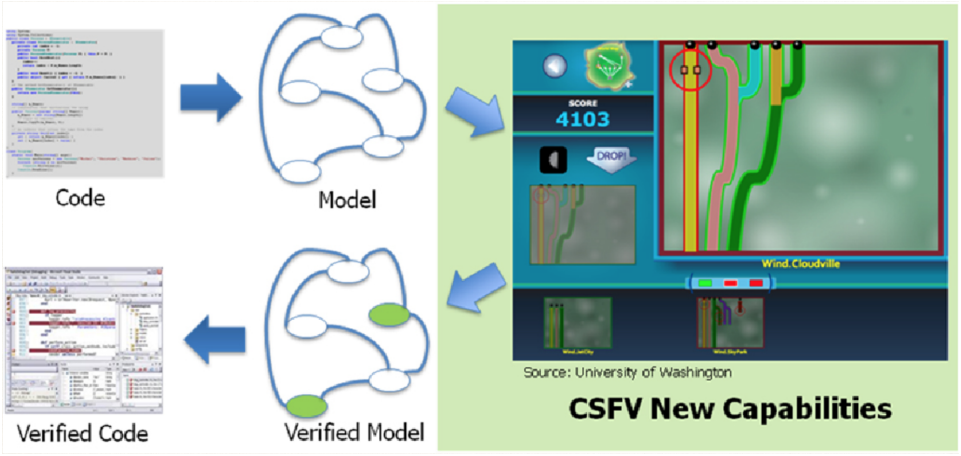


*Figure 2.1: Crowd source formal verification*

A particular game instance is a function of the program verification tool, the property to be verified and the program being verified. Each game instance is released to the crowd, either via the Web or through internal domain distribution. Game solutions collected in this way are then used to populate a database.
A reverse mapping is done to insert back into a program annotations sufficient to allow a verification tool to make progress toward verifying a specific program property.

The process of rigorously analyzing software to detect flaws that make programs vulnerable to exploitation requires highly skilled engineers with extensive training and experience. This makes the verification process costly and relatively slow. An example of a real world ap-

plication of CSFV is the Verigames game collection[25]. Verigames seeks to replace the intensive work done by the domain experts by greatly decreasing the skill required to do Formal Verification, and therefore allow more people (who do not need to be domain experts) to perform the analysis in a more efficient manner. By creating fun and engaging games that represent the underlying mathematical concepts, they empower the non-experts to effectively do the work of the formal verification experts simply by playing and completing the game objectives.


*Figure 2.2: Verigames*

For instance in the game *Xylem*[26] the player is an experienced botanist with the goal of identifying and cataloging Miraflora's plant life using skills in observation and problem-solving. The player has to insert information about the growth phase of flower species that are true in every growth phase shown in the game instance. An example of a particular game instance is shown in Figure 2.3.


*Figure 2.3: Xylem Game Instance*

Here the the number of flowers of the two specimens are related by a particular equation. The player may notice that the growth of the flower follows a sum of arithmetic progression terms $Sn = \frac{n}{2(2a1+(n-1)*d)}$ and submit it as shown in Figure 2.4.

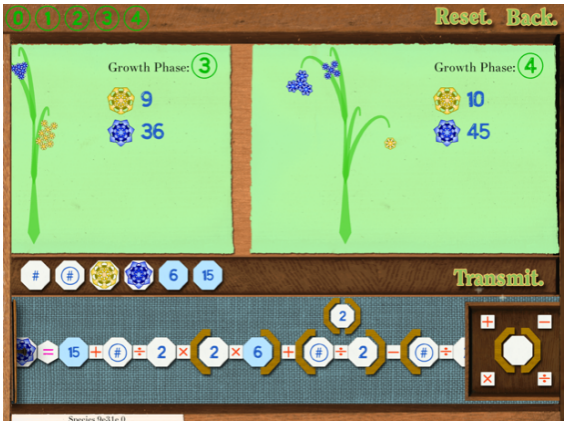


*Figure 2.4: Xylem Game Solution*

Eventually a feedback about the quality of his solution is shown to the player that gets his reward in terms of game points as in Figure 2.5.

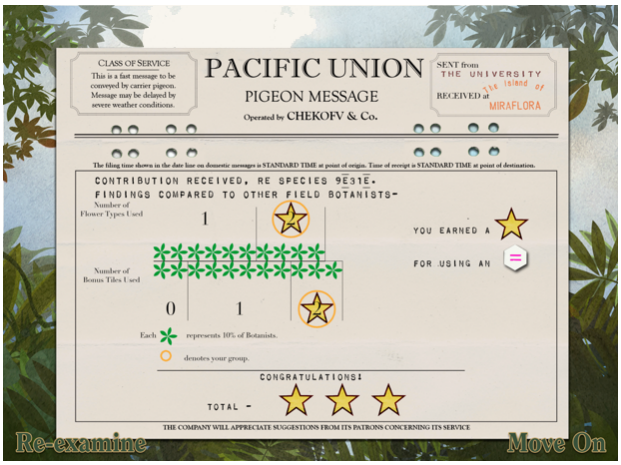


*Figure 2.5: Xylem Game Feedback*

### 2.2.3 Program Analyzers

Program analyzers automatically analyze the behavior of programs. They are able to generate informations that can be automatically inserted in the program SC or IR. Different program analysis, such as weakest precondition calculus and value analysis, can be helpful both for testing the correctness of a program and for optimizing it. In addition, these tools can aid developers debugging.

The input program is usually parsed using a custom-built front end that performs pointer analysis, heap modeling, slicing, constant folding and numerous simplifications to construct a Control Flow Graph (CFG) model of the program. Program analyzers may exploit different theories in order to accomplish the task of verifying a certain property or obtaining a certain information. The two main theories on which the most of the program analyzers are based are *Abstract Interpretation* and *Lazy Abstraction*. In the next subsections follows a brief description of them.

#### Abstract Interpretation

Abstract Interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets (such as lattices). It can be viewed as a partial execution of a computer program which gains information about its control-flow and data-flow without performing all the calculations. *Symbolic Execution*, a specific case of *Abstract Interpretation*, analyze a program to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

#### Lazy Abstraction

Lazy abstraction is a theory based on the abstract-check-refine paradigm. It consist of three phases: build an abstract model, then check the desired property, and if the check fails, refine the model and start over.

It basically cycles through the following loop:

1. *Abstraction* a finites et of predicates is chosen, and an abstract model of the given program is built automatically as a finite or push-down automaton whose states represent truth assignments for the chosen predicates.

2. *Verification* The abstract model is checked automatically for the desired property. If the abstract model is error-free, then so is the original program (the correctness proof ends) otherwise, an abstract counterexample is produced automatically which demonstrates the property violation.

3. *Counterexample-Driven Refinement*: It is checked automatically if the abstract counterexample corresponds to a concrete counterexample in the original program. If so, then a program error has been found (then the program proven incorrect) otherwise, the chosen set of predicates does not contain enough information for proving program correctness and new predicates must be added.

4. *Loop-Back*: go to step 1.

*Lazy abstraction* continuously builds and refines a single abstract model on demand, driven by the model checker, so that different parts of the model may exhibit different degrees of precision, namely just enough to verify the desired property.

C program verification tools such as BLAST[29], Frama-C[28], F-Soft[27] are based on these theories and employ SMT solvers to produce precise (inductive) invariants. In Chapter 3 we will describe *Aruna* , a framework that uses the invariants coming from Frama-C Value Analysis (that relies on *Symbolic Execution*) in order to improve the compilation process.

## 2.3 Annotation Languages

By *annotation languages* we mean program behavioral specification languages. An annotation language should be able to express a wide range of functional properties.

As an example Java has a syntax[31] for declaring annotation types, a syntax for annotating declarations, APIs for reading annotations, a class file representation for annotations and an annotation processing tool.

As we said before, these annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at run time. When Java source code is compiled, annotations can be processed by compiler plugins called annotation processors. Processors can produce informational messages or create additional Java source files or resources, which in turn may be compiled and processed, and also modify the annotated code itself. The JVM or other programs can look for the metadata to determine how to interact with the program elements or change their behavior.
Annotation languages can be generally classified by their purposes:

- *Organization Languages*: they can be used by the programmers to better organize a code base. For example the *#pragma mark* directives in Objective-C is useful in helping the XCode IDE in organizing the class methods. When the programmer has an application with classes with more the ten method definitions, it is a very good idea to use *#pragma mark* directives so that the code looks neater and more organized.

- *Syntactic Languages*: they can be used to mark some portion of the source code in order to help the syntactic checks and reduce programmer's errors. An example can be the Java annotations @Deprecated, @Override and @SuppressWarnings.

- *Specification Languages*: they can be used to specify properties about the code. Specification languages are generally not directly executed. They are meant to describe the *what*, not the *how*. Usually there is a broad range of annotations that can be specified with those languages. For instance the ANSI/ISO C Specification Language (ACSL) language supports in-function annotations (assertions, loop invariants, ghost code, ...), function annotations (preconditions and postconditions) and global annotations (predicates, logic functions, type and global invariants, ...).

- *Specific Application Languages*: they are less general and are used to annotate the code so as to help a specific program analyzer or verification tool.

Standard annotation languages such as the ACSL[13] for C and the Java Modeling Language (JML)[18] for Java are widely used during software

development. In addition annotation languages offer a standard information representation that can be also used by program analyzers both as an input to verify properties and as an output to insert additional information in the code. The use of a standard annotation language in our framework makes it easier to support a broad range of annotations by means of a well defined specification. Since *Aruna* targets C programs it will rely on the ACSL language as an interface for the input invariants coming from external analyzers.

## 2.4 Compiler Architecture

A typical compiler structure is composed of two subsystems, the *front-end* and the *back-end*. This structure is shown in Figure 2.6.
The intermediate representation is independent of the specific source or machine languages and acts as an interface between the front-end and back-end.

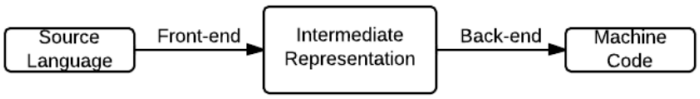


*Figure 2.6: Typical compiler structure*

The major advantage of this split is that it is easier to design back-ends that are independent of input source language and vice-versa for front ends with respect to machine properties: suppose there are N target source languages and M target machines. This approach allows for $N + M$ (front-end , back-end) pairs instead of $N \times M$ whole compilers for every (source language, machine) pair. The goal of our framework is to exploit the same principle to be more general and reusable as possible between different annotation sources and back-end optimizations. The choice of a standard annotation language discussed in Section 2.3 will act as an interface, as the IR does between the back-end and the front-end. The goal of our framework will be to be the bridge between the annotation in the SC and IR used in the back-end. The detail about the propagation architecture is shown in Chapter 3.
While reading this dissertation you will find lots of C SC to LLVM IR examples. The basic LLVM IR concepts are highlighted in Section 2.4.1. Section 2.4.2 describes the basic modules of the LLVM architecture used in our framework.

### 2.4.1 Static Single Assignment Form

The Static Single Assignment (SSA) form[19] is a property of an intermediate representation, where each variable is assigned exactly once. Existing variables in the original IR are split into multiple variables. These new variables typically indicated by the original name with a subscript in textbooks[17], so that every definition gets its own version. However, as we will see in later code examples, in LLVM they usually take the name of the operation that is performed. This form usually simplifies data-flow analysis and program optimizations and reduces the space and time complexity needed while following def-use chains.

In Listings 2.5, 2.6 and 2.7 follow a simple example of code translated into SSA form.

*Listing 2.5: SSA Example*

```
...
x = y * z;
y = x + 3;
x = y + 4;
z = y * 5;
x = x + z;
...
```

*Listing 2.6: SSA Example Textbook IR*

```
...
x1 = y1 * z1;
y2 = x1 + 3;
x2 = y2 + 4;
z2 = y2 * 5;
x3 = x2 + z2;
...
```

*Listing 2.7: SSA Example LLVM IR*

```
...
%mul = mul nsw i32 %y, %z
%add = add nsw i32 %mul, 3
%add1 = add nsw i32 %add, 4
%mul2 = mul nsw i32 %add, 5
%add3 = add nsw i32 %add1, %mul2
...
```

Usually compilers first convert the program into an IR SSA form, then perform the optimization passes and eventually they translate the IR into machine code.

### 2.4.2 LLVM

Since in our implementation we are using program written in C as benchmarks we choose to target the LLVM Architecture[2][10]. The LLVM Project is a collection of modular and reusable compiler tools. In this work we show how we modified both the LLVM Core and the SAFECode project (that is built using the LLVM compiler infrastructure) so to reduce the trade-off between security and execution time of a compiled program and improve current compiler optimizations.
Here it follows a brief description of the main LLVM tools and concepts used in *Aruna* :

- *Clang*[14] is C, C++, Objective C and Objective C++ front-end for the LLVM compiler. It can be use to emit LLVM IR that can be later used to optimize and compile the code.

- *LLVM IR*[7] is a SSA based representation that allows many source languages to be mapped to them. It is the common code representation used throughout all phases of the LLVM compilation strategy and acts like an interface between the LLVM Core Passes.

- *LLVM Core*[20] are a set of libraries that provide a modern source-independent and target-independent optimizer, along with code generation support for many CPUs. These libraries are built around the LLVM IR. Optimizations are implemented as Passes[11] that traverse some portion of a program (such as functions, loops and basic blocks) to either collect information or transform the program.

- SAFECode[6] project is a memory safety compiler built on top of LLVM. It is used to prevent the compiled program from having security flaws so as to protect software from security attacks. Specifically it instruments code with run-time checks to detect memory safety errors (e.g. buffer overflows) at run-time.

# Chapter 3

# Approach

This chapter describes the *Aruna* framework architecture and some of its applications showing how the tool can be used to help the compilation process.

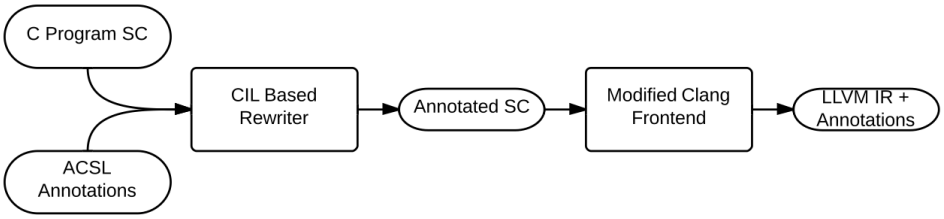


*Figure 3.1: Aruna  Propagation Framework Architecture*

The basic idea behind our approach is to instrument the program source with annotations. This is done to facilitate the processing of annotations by the rest of the compiler toolkit. Figure 3.1 shows the basic architecture of *Aruna* . The input to the tool is a C source program and a list of assertions produced by any static analysis tool. These annotations are expressed in ACSL, a specification language that supports a wide variety of annotations. A custom CIL-based[30] rewriter visits each instruction and injects annotations specific to that instruction before and/or after the instructions, resulting in a C source code annotated with assertions.

The annotated source is then passed on to Clang, the front-end of the LLVM compiler, whose role is to translate the C source into LLVM IR. *Aruna* augments Clang so to bind the assertions of the C source into the LLVM IR. Obviously, the annotations should appear in the right locations and refer to the right variables.

There are several challenges in accomplishing this, including:

- *Language Heterogeneity*. The annotations are produced for the C source, and are to be mapped into the LLVM IR so to allow their use by LLVM. This requires careful manipulation of the assertions.
- *Scope Resolution* The annotations may appear in a specific scope and use variables that are alive in a different scope, therefore scope resolution across different basic blocks in the LLVM IR is also required.
- *Information Preservation*. As the compiler performs its passes, the assertions captured by the annotations need to be modified to reflect the changes in the IR. *Aruna* has to propagate the annotations so that they always carry all the correct and relevant information.

These challenges are addressed by a *Variable Mapping Pass* that run immediately run after the first Clang translation to LLVM IR (we can see it the final step of the front-end phase). This pass translates the annotations referring to the source code to annotations referring to the LLVM IR, so that they can be finally used by the general optimization framework of LLVM. In Sections 3.3.2 and 3.3.3 we give an overview of these framework applications.

## 3.1 Goals and challenges

We seek to demonstrate how the annotations in the source code can be used during the compilation process and how to achieve better code performances by relying on them. We are not concerned with time of the compilation process. Of course, the generation of code with annotation can take time and analysis algorithm can have an high complexity, however here we are only focusing in taking advantage of already annotated code (that can come from different sources as seen in Section 2.2).
In order to test our approach we used the formal verification tool for C programs Frama-C[3]. In particular we use the results of Frama-C's *Value Analysis*[4] plug-in and embedding these results in the source code. To make our framework more general and highly reusable we are supporting annotations written in the standard ACSL, so that our back-end can use also information from different kind of analyses or sources.

In addition we show how these achievements can be obtained in a real world state-of-the-art compiler such as LLVM and in some real-world C programs described in Section 4.1.2. We will see in detail in Chapter 4 how we are facing the challenge of plugging our annotation in the IR in order to make them useful for optimization Passes.

## 3.2 Overview

The goal of the *Aruna* framework is to obtain a compiler that performs optimizations based on high-precision analysis obtained with tools for static analysis and formal verification. We are willing to enhance two aspects of the compilation process: program security and performance. In the next subsections we present an overview of the approach taken to address these goals.

### 3.2.1 Lightweight Run-time Checks Injection

In order to strengthen security, the LLVM SAFECode Project is designed to prevent pointers from overflowing from one memory object into another by inserting run-time array bounds checks into the program code. This prevents buffer overflows, one of the major *mitre25* [12] vulnerabilities. However, this comes at the cost of a trade-off between security and code performance. In Section 3.3.2 we show how we used the information coming from program annotation to reduce the cost of this trade-off.

### 3.2.2 Pushing Current Optimizations Forward

For efficiency's sake, the existing optimizations rely on mostly linear, rarely quadratic, analysis algorithm. This is especially true with the advent of just-in-time compilation. For example in the GCC Developer Wiki [15] there is specified not to add algorithms with quadratic or worse behavior. Since we are relying on existing additional information that can be generated by more powerful kind of analysis, we believe that this information can be used for new optimizations or can improve already existing ones. We will focus on modifying current LLVM optimizations.

## 3.3 Details

In order to evaluate *Aruna* and to show some its applications we collected some annotated code by means of running the *Frama-C* [3] analyzer on some benchmark C programs.
Since the LLVM back-end compilation process is accomplished by a sequence of passes iterations, in both the application of our framework the compilation starts after a first annotation mapping pass that is run before the normal passes sequence so as to make the annotations meaningful and available to the later compilation passes. The following subsections generally illustrates the basic modules of our two framework applications and how they are chained together to achieve the desired improvements.

### 3.3.1 Annotated Programs Generation

*Frama-C* has a plug-in called *Value analysis* [4] that computes variation domains for variables. This plug-in uses abstract interpretation techniques and it handles a wide spectrum of C constructs. The Frama-C graphical user interface displays the inferred sets for possible values of a variable in each point of the analyzed program. This plug-in give us information about variables before and after the selected line of code.
A custom Frama-C plug-in captures the result from *Value analysis* plug-in, log them in a separate file and then reinsert them in the original C source code as ACSL annotations.

### 3.3.2 Lightweight SafeCode Pipeline

Since we need to associate each information about index of array accesses to the correct array access instructions, we created the pipeline of passes showed in Figure 3.2. Information about the index in array accesses might allow us to remove the check if the variable is always in bounds.

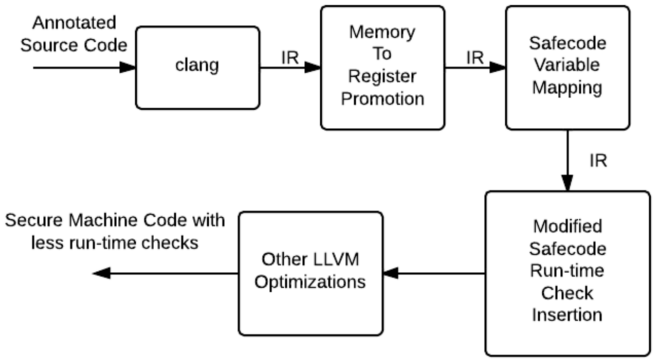


*Figure 3.2: Lightweight SafeCode Pipeline*

After the memory to register promotion a pass will map the array index in the source code to the corresponding register in the IR. Then it will attach a metadata with the information about the range of the register used for the access (if any) to the instructions used for the access. The later SafeCode passes are then modified so that these information are used to the inspect each injection and to decide if the checks are really needed or not.

### 3.3.3 LLVM New Optimizations pipeline

In order to push the information in the source code annotations throughout the compiler architecture, we designed a pipeline of steps that is showed in Figure 3.3.

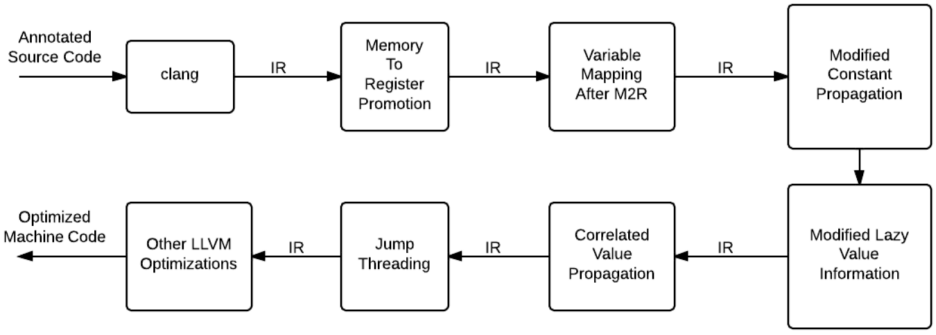


*Figure 3.3: LLVM new optimizations pipeline*

The purpose of this pipeline can be summarized in the following steps:

- *From source code to IR*
  The annotations placed in the source code as strings are translated by *clang* into the IR as global constant strings. An example is showed in Listing 3.1.

  *Listing 3.1: String Annotation Example*

```
1 @.str = private unnamed_addr constant [15 x i8] c"@assert␣k␣==␣
      1\00", align 1
```

  For each instruction Clang inserts a debug information that is later used during the following pipeline steps.

- *Memory to Register*
  The string declared and defined in the source code is removed after the memory to register promotion, hence we are justified in ignoring code size effects of the choice. However, the debug information is preserved at the same line of code where the strings were declared and defined, hence we still have additional information in the IR ready to be used.

- *Mapping Annotation Variables to Registers*
  The annotations contain information about source code variables, however at this stage of the pipeline the source code variables are already associated to SSA registers. Our pass scans the debug information in the IR and performs and update the annotation variables accordingly to the correct mapping.

- *Modified Constant Propagation*
  We modified the LLVM Constant Propagation Transformation Pass so it depends on constant information from annotated source code.

- *Modified Lazy Value Informations*
  We modified the LLVM Lazy Value Info Analysis Pass so to consider constant and range information from the annotated source code.

- *Run optimizations dependent on value analysis*
  We run both Correlated Value Propagation and Jump Threading since that depend on the modified Lazy Value Info Analysis.

- *Other LLVM Optimizations*
  We strip the all the debug information and run the "normal" LLVM

optimizations to obtain the optimized code. These later optimizations can also benefit by the improved modified optimizations and make more effective changes.

The choice of the passes order is a constraint due to the propagation of the annotation information throughout the transformation passes. In Chapter 6 we describe how to remove this constraint.

# Chapter 4

# Implementation

The goal of our framework *Aruna* is to propagate invariants coming from external annotation sources during the LLVM front-end compilation so as to make them available to the back-end passes. In order to be ready for future applications *Aruna* relies on a ACSL flex/bison parser wrapped in a C++ driver that can be easily extended to handle complex annotations. *Aruna* is built on top of the LLVM architecture and can be used both with the normal LLVM installation or with other projects built on top of LLVM. This is why we evaluated our framework by using the external information coming from the Frama-C program analyzer both in the normal LLVM compilation process and in the SafeCode project compilation.
In this chapter we are going to present the most interesting details about the implementation of this work. We are going to explain how the annotated code was generated, the different components of the SafeCode and optimization pipelines and some insights about how the engineering hurdles where handled.

## 4.1 Annotated Code Data Collection

Here we present how we are automatically generating some C programs with annotations to test our approach. The framework can take every program already annotated (also manually) and trust the additional information coming from the annotations to improve the existing optimizations. In our application we rely on the external invariants proven by Frama-C. In subsection 4.1.2 we present an overview of the benchmark programs used to evaluate the framework applications.

### 4.1.1 Frama-C Annotations

Frama-C annotations are only available via the GUI interface. To bring these information into the source code, we implemented a Frama-C plug-in to perform the work.

This plug-in visits every assignment and function call instruction in the AST tree in Frama-C, extracts all variables and queries the value analysis plug-in for each variable to get the possible values. In this work we are only interested in constant and range bounds of a variable, we ignored other information of complex variables such as structs, arrays or pointers. Since the Clang front-end discards comments, *Aruna* needs to use an annotation format that is not removed by Clang. This information is inserted as string in ACSL language into the C source code via a dummy string variable before and/or after the inspected instruction whenever the information from the Value Analysis plug-in is available. The variables are specially named so that they do not interfere with the existing program variables. As the annotations are encoded as assignments to special variables, these annotations are propagated to the LLVM IR by the Clang front-end compiler.

Frama-C merges multiple file into a single file. This will change the multiple file programs structure and might causes compiling issues of large programs. In order to handle multiple files programs we log the value information into a file together with the location and type of instructions in original source file and then we inject the dummy string variable matching the location and type of an instruction stored in the log file via a custom CIL[21] plugin. Our CIL-based rewriter visits nodes in the abstract syntax tree of program, and injects the provided annotations as input into the corresponding nodes.

### 4.1.2 Benchmarks used

In order to evaluate this work we annotated some C programs using the custom plugin mentioned in Section 4.1.1. Here it follows a short description of the benchmarks used:

- CoreMark is a benchmark that aims to measure the performance of CPU used in embedded systems. The code contains implementations of list processing (find and sort) and matrix manipulation (common matrix operations) algorithms.

- SUSAN is a benchmark that implements algorithms based on Smallest Univalue Segment Assimilating Nucleus. The SUSAN algorithms cover image noise filtering, edge finding and corner finding.
- MxM is a benchmark that computes matrix-matrix products in multiple different ways.
- Linpack is a benchmark that implements algorithms for vector sum, vector product, scaling vectors by a constant, matrix factorization, solving linear systems and random number generation.
- NEC-Matrix is a small benchmark that contains the implementation scalar product and some multiplication and addition over matrices.

## 4.2 Pushing the annotations through the CLANG front-end

The annotations inserted in the code as ACSL comments are ignored and removed by the clang C front-end. Therefore, in order to keep this information in the very first stages of the back-end compilation process, the annotations are inserted in the C program as C strings. This choice was made because since these strings are just dead code they will be easily removed during later optimizations, hence it will not affect both the code size of the output program and its performances.

## 4.3 The ACSL Parser

To the extent of handling different kinds of annotations and parse them we wrote an ACSL parser that builds an Abstract Syntax Tree (AST) out of every annotation string in input. For our purposes we are supporting only a small set of annotation, however the grammar can be easily extended to support a broader variety of ACSL constructs.

### 4.3.1 ACSL Supported Subset

We are currently supporting preconditions (*@requires*), postconditions (*@ensures*) and assertions (*@assert*) containing boolean expressions about variable values. The supported grammar is showed in Appendix A.

In Listing 4.1 we present a simple assertion about variable ranges and constant values:

*Listing 4.1: Supported ACSL Example*

```
1      @assert i>=0 && i<=10   assert j==0     assert k==1 || k==2
```

For the sake of this work this ACSL subset is sufficient, however the parser can be easily extended to support new annotations coming from other sources.

### 4.3.2 Parser Generation

The ACSL parser is an automatically generated using *Flex* (a scanner generator) and *Bison* (a parser generator) tools. The reason to use these tools is that the code generated by them requires no compile-time dependencies, because they generate fully autonomous source code. In addition we do not need to rewrite all the ACSL parser but we only need to modify the files used from these tools to handle new ACSL constructs. Therefore it will be really easier to augment the parser just by learning how to use these common tools.

The output from the *Flex* scanner and *Bison* parser pair is encapsulated into classes in order to incorporate it into a modern C++ program as LLVM. Precisely the class that which puts together lexer and parser is the *Driver* class. The *Driver* class is independent from the automatically generated files and exposes methods to get the AST given as an input a string, a file or a stream.

### 4.3.3 The ACSL AST

The diagram of the classes that compose the ACSL AST are shown in Figure 4.1.

All the classes in the diagram implement the LLVM style Run-Time Type Information (RTTI) that can be used to runtime check the generated structure of the AST. In addition the *ACSLNode* class has a few methods useful for extracting the list of variables in the parsed annotation and to rename them (we will see in Section 4.4 why we need variable renaming).

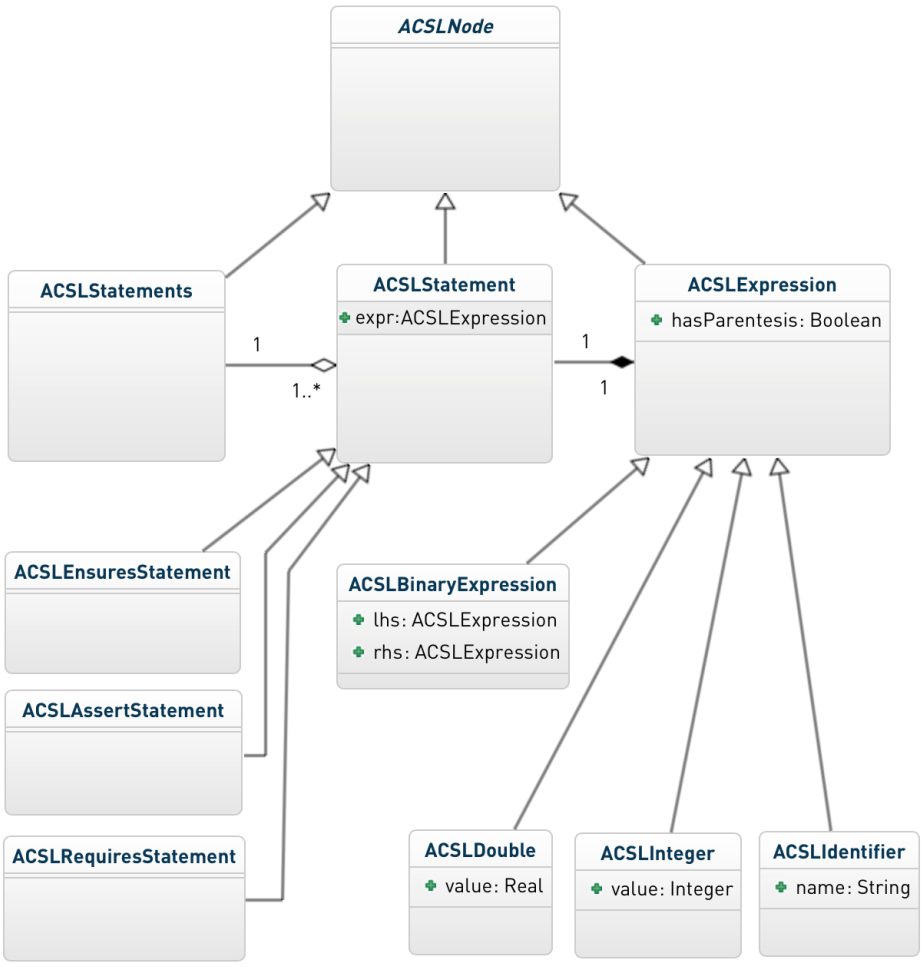


*Figure 4.1: ACSL AST class diagram.*

## 4.4 Mapping Source Code Variables to IR Variables

During the front-end compilation into LLVM IR every source code variable declaration is associated a memory location. Every later access to that variable is done via load and store instruction. However if we have a nested scope with a variable name equal to the variable name in a scope on top of that the two memory location will get two different names. In Listings 4.2 and 4.3 we show the issue with a trivial example.

*Listing 4.2: Same Name Different Scope Example*

```
int main() {
        //this will be named %x
        int x = 0;
        char * a1 = "@assert x==0";
        if ( x >= 0 ){
                //this will be named %x1
                int x = 1;
                char * a2 = "@assert x==1";
        }
        return x;
}
```

*Listing 4.3: Same Name Different Scope Example IR*

```
@.str = private unnamed_addr constant [13 x i8] c"@assert x==0\00"
    , align 1
@.str1 = private unnamed_addr constant [13 x i8] c"@assert x==1\00
    ", align 1
define i32 @main() nounwind ssp uwtable {
        %1 = alloca i32, align 4
        %x = alloca i32, align 4
        %a1 = alloca i8*, align 8
        %x1 = alloca i32, align 4
        %a2 = alloca i8*, align 8
        store i32 0, i32* %1
        store i32 0, i32* %x, align 4
        store i8* getelementptr inbounds ([13 x i8]* @.str, i32 0,
            i32 0), ...
        %2 = load i32* %x, align 4
        %3 = icmp sgt i32 %2, 0
        br i1 %3, label %4, label %5
        ; <label>:4                                     ; preds
            = %0
        store i32 1, i32* %x1, align 4
        store i8* getelementptr inbounds ([13 x i8]* @.str1, i32
            0, i32 0), ...
        br label %5
        ; <label>:5                                     ; preds
            = %4, %0
        %6 = load i32* %x, align 4
        ret i32 %6
}
```

In order to map the names of the identifiers in our annotations to the correct names in the IR the *ACSLVarMap* Pass maps the variable to the correct name using debug information inserted by Clang in the IR[8] (by running clang with the -g argument). This pass is useful for every optimization that runs before memory to register promotion. In Section 4.4.1 we show how the mapping is done.

The most effective LLVM optimizations run after the *PromoteMemoryToRegister* Pass. This pass promotes memory locations to registers in SSA Form and inserts $\phi$ functions. In order to use the information coming from the annotations the *ACSLVarMapAfterM2R* Pass maps every variable to the correct instruction name in the LLVM IR. In Section 4.4.2 we show how the mapping is done.

### 4.4.1 ACSL Variable Mapping Pass

The *ACSLVarMap* Pass implements an algorithm that decodes the debug information inserted by the front-end and uses them to map the variable names in the source code to the correct memory locations in the IR. This algorithm iterates three times over the function body.
Here follows the implementation details of the two loops:

- *Naming Instructions Without Names:*
  Since our implementations relies on instruction names to map variables in the annotations a first loop assign to unnamed instruction a fresh unique name.

- *Gathering Debug Informations:*
  To every allocation instruction (resulting in the memory location of the variable) corresponds a call to the llvm.dbg.declare function. The signature *is void %llvm.dbg.declare(metadata, metadata)*. This intrinsic provides information about a local element (e.g., variable): the first argument is metadata holding the allocation for the variable, the second argument is metadata containing a description of the variable. In Listing 4.4 we show a simple example of how it is translated simple *int x = 0;* C statement.

*Listing 4.4: llvm.dbg.declare Example*

```
1       %x = alloca i32, align 4
2       store i32 0, i32* %x, align 4, !dbg !19
3       ...
4       call void @llvm.dbg.declare(metadata !{i32* %x}, metadata
            !18), !dbg !19
5       ...
6       !18 = metadata !{i32 786688, metadata !5, metadata !"x", ...
7       !19 = metadata !{i32 7, i32 0, metadata !5, null}
```

  In this first loop that iterates over the instructions in every function, we store in a data structure all the information about:

  - Source Code Name
  - IR Name
  - Source Code Scope

  This mapping information will be used in the following loop.

- *Collecting and Mapping Annotations:*
  To every string annotation in the source code corresponds an allocation instruction, then a store with the corresponding string con-

tent. In Listing 4.5 we show a simple example of how it is translated a simple *char * a = "@assert y==100"* C statement.

*Listing 4.5: String Annotation Example*

```
1  @.str = private unnamed_addr constant [15 x i8] c"@assert y
       ==100\00", align 1
2  %a = alloca i8*, align 8
3  ...
4  call void @llvm.dbg.declare(metadata !{i8** %a}, metadata
       !22), !dbg !25
5  ...
6  store i8* getelementptr inbounds ([15 x i8]* @.str, i32 0,
       i32 0), i8** %a, align 8, !dbg !25
```

Once we get the string content we parse it using the *Driver* Class of the ACSL Parser. On the AST we get the list of the variables in the annotations calling the *getTreeVariables* method on the root of the AST. For every variable we solve the mapping using the data structure created in the first loop.
In order to get the correct mapping we iteratively try to solve the variable mapping in the same scope of the annotation, then if we don't find a candidate we start over in the outer scope of the current one.
Finally we substitute the correct names in the AST calling the *changeTreeVariableName* method. This annotations are then attached as a string *acsl_metadata* to the corresponding IR instructions.

### 4.4.2 Handling Memory to Register Promotion

The LLVM *PromoteMemoryToRegister* Pass converts allocations to registers. An allocation is transformed by using iterated dominator frontiers to place $\phi$ nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. It also propagates the constant value of declarations immediately followed by a definition (i.e. *int x = 0;*).
A simple example of the IR produced after the *PromoteMemoryToRegister* Pass is showed in Listings 4.6 and 4.7.

*Listing 4.6: Memory to Register Example*

```
1   int main() {
2
3           int y=100;
4           int x=0;
5
6           char * a1 = "@assert␣x==0";
7           x = y + 1;
8           char * a2 = "@assert␣x==101";
9
10          return x;
11  }
```

*Listing 4.7: Memory to Register Example IR*

```
1   ...
2   @.str = private unnamed_addr constant [13 x i8] c"@assert␣x==0\00"
        , align 1
3   @.str1 = private unnamed_addr constant [15 x i8] c"@assert␣x
        ==101\00", align 1
4
5   define i32 @main() nounwind ssp uwtable {
6   entry:
7     %add = add nsw i32 100, 1
8     ret i32 %add
9   }
```

In order to correctly map the identifiers in our annotations to the correct registers the *ACSLVarMapAfterM2R* Pass should be run immediately after the *PromoteMemoryToRegister* Pass. The implemented algorithm iterates three times over the function body.
Here follows the implementation details of the four loops:

- *Naming Instructions Without Names:*
  As we have seen before since our implementations relies on instruction names to map variables in the annotations a first loop assign to unnamed instruction a fresh unique name.

- *Gathering Debug Informations:*
  The debug information is similar to the one in Section 4.4.1. The difference is that in addition to the llv.dbg.declare calls we are also keeping track of the llvm.dbg.value calls. The signature is *void %llvm.dbg.value(metadata, i64, metadata)*. This intrinsic provides information when a user source variable is set to a new value. The first argument is the new value (wrapped as metadata). The second argument is the offset in the user source variable where the new value is written. The third argument is metadata containing a description of the user source variable. The example in Listing 4.6 after the *PromoteMemoryToRegister* Pass with debug information is showed Listing 4.8.

*Listing 4.8: Debug Information in Memory to Register Example IR*

```
1       ...
2       @.str = private unnamed_addr constant [13 x i8] c"@assert_x
            ==0\00", align 1
3       @.str1 = private unnamed_addr constant [15 x i8] c"@assert_x
            ==101\00", align 1
4
5       define i32 @main() nounwind ssp uwtable {
6       entry:
7         ;these debug information is about y and x
8         call void @llvm.dbg.value(metadata !10, i64 0, metadata
              !11), !dbg !12
9         call void @llvm.dbg.value(metadata !2, i64 0, metadata
              !13), !dbg !14
10
11        ;this debug info is about the first annotation
12        call void @llvm.dbg.value(metadata !15, i64 0, metadata
              !16), !dbg !19
13
14        %add = add nsw i32 100, 1, !dbg !20
15
16        ;this debug information is about x
17        call void @llvm.dbg.value(metadata !{i32 %add}, i64 0,
              metadata !13), !dbg !20
18
19        ;this debug info is about the second annotation
20        call void @llvm.dbg.value(metadata !21, i64 0, metadata
              !22), !dbg !23
21
22        ret i32 %add, !dbg !32
23      }
24      ...
25      !2 = metadata !{i32 0}
26      ...
27      !10 = metadata !{i32 100}
28      !11 = metadata !{i32 786688, metadata !5, metadata !"y",
            metadata !6, i32 3, metadata !9, i32 0, i32 0}
29      !12 = metadata !{i32 3, i32 0, metadata !5, null}
30      !13 = metadata !{i32 786688, metadata !5, metadata !"x",
            metadata !6, i32 4, metadata !9, i32 0, i32 0}
31      !14 = metadata !{i32 4, i32 0, metadata !5, null}
32      !15 = metadata !{i8* getelementptr inbounds ([13 x i8]* @.
            str, i32 0, i32 0)}
33      ...
34      !21 = metadata !{i8* getelementptr inbounds ([15 x i8]* @.
            str1, i32 0, i32 0)}
35      ...
```

In each iteration of this first loop we store in a data structure all the information about:

- Source Code Name
- IR Name
- Source Code Scope
- Basic Block
- Line of Code

This mapping information will be used in the third loop.

- *Handling $\phi$ functions:*
  Registers associated to $\phi$ functions have no debug information (since

they are not in the original code). However they still are important because they can be the target name of a variable in our annotations.
In order to handle $\phi$ functions we should update the data structure with other mapping information. The source code name is obtained by looking at the arguments of the $\phi$ function. Then we add the information in the data structure at the next line of code. In addition if the source code variable is not redefined in the current basic block we push at the beginning of the successors of the current basic block the information about the analyzed $\phi$ function in the data structure.

- *Collecting and Mapping Annotations:*
  As we can see in Listings 4.6 and 4.7, the allocation and store instruction that we were able to use before the *PromoteMemoryToRegister* Pass to catch the annotations strings are no longer in the IR. Fortunately as we can see in Listing 4.8 we still have a *llvm.dbg.value* call for each annotation string.
  The retrieve annotation will be parsed as in Listing 4.4.1. The only difference is in the algorithm to get the correct IR name mapping. We will search backward (looking for a smaller line number) if there is a correct mapping information in the same scope and basic block, if not, we will carry on backward searching in the predecessor to the current basic block until we find a candidate.

## 4.5 Reducing SafeCode Checks

The LLVM BackendUtil class is modified to instrument the pipeline of passes to be run before the already existing SafeCode Passes as previously showed in Figure 3.2. Using a modified version of the *ACSLVarMapAfterM2R* Pass that we called *SafecodeVarMap* Pass we where able to add metadata information to the GetElementPtr (GEP), load and store instructions. This information can be later used in the later SafeCode Passes to avoid to check formally proven secure accesses.

### 4.5.1 Adding information to GEP, LOAD and STORE instructions

If a variable that is inside an annotation is later used in the same basic block as an operand of a GEP, LOAD or STORE instruction, to each

of these instructions will be attached a named *acsl_safecode* metadata containing the annotation.
A simple example is showed in Listings 4.9 and 4.10.

*Listing 4.9: Array Access Example*

```
1   int main() {
2           int a[10];
3           int i=0;
4           char * a1 = "@assert␣i==0";
5           for( i = 0 ; i < 10 ; i++ ){
6               char * a2 = "@assert␣i>=0␣&&␣i<10";
7               a[i]=i;
8           }
9           return i;
10  }
```

*Listing 4.10: Array Access Example IR*

```
1   @.str = private unnamed_addr constant [13 x i8] c"@assert␣i==0\00"
        , align 1
2   @.str1 = private unnamed_addr constant [21 x i8] c"@assert␣i>=0␣&&
        ␣i<10\00", align 1
3
4   define i32 @main() nounwind ssp uwtable {
5   entry:
6     %a = alloca [10 x i32], align 16
7     call void @llvm.dbg.declare(metadata !{[10 x i32]* %a}, metadata
          !10), !dbg !14
8     ...
9     br label %for.cond, !dbg !22
10
11  for.cond:                                         ; preds = %for.
        inc, %entry
12    %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
13    %cmp = icmp slt i32 %i.0, 10, !dbg !22
14    br i1 %cmp, label %for.body, label %for.end, !dbg !22
15
16  for.body:                                         ; preds = %for.
        cond
17    call void @llvm.dbg.value(metadata !24, i64 0, metadata !25), !
          dbg !27
18    %idxprom = sext i32 %i.0 to i64, !dbg !28
19    %arrayidx = getelementptr inbounds [10 x i32]* %a, i32 0, i64 %
          idxprom, !dbg !28
20    store i32 %i.0, i32* %arrayidx, align 4, !dbg !28
21    br label %for.inc, !dbg !29
22
23  for.inc:                                          ; preds = %for.
        body
24    %inc = add nsw i32 %i.0, 1, !dbg !22
25    call void @llvm.dbg.value(metadata !{i32 %inc}, i64 0, metadata
          !15), !dbg !22
26    br label %for.cond, !dbg !22
27
28  for.end:                                          ; preds = %for.
        cond
29    ret i32 %i.0, !dbg !30
30  }
```

We are supporting accesses to arrays of fixed size created using statements of the type "*int array[10];*" and "*int * array = malloc(10 * sizeof(int));*" of any type.

In addition, if the index of the access is the result of an expression (for example "*array[i*j+2]*"), we are currently propagating the ranges information to the SSA registers following the annotation in the same basic block. This is done using simple interval operations described in Appendix B.

The same is done for array whose size is the result of an expression (for example "*int array[i*j]*"). The minimum size coming from the range information propagation is used. This means that we are not injecting the check only if the array index is in bound considering the smallest size.

We are also keeping into account the sign extension of integer types (SEXT) in order to attach the annotation to the correct GEP, LOAD or STORE instruction. The sign extension is needed for example when we are using in 32bit integer to access a 64bit indexed array.
The resulting modified IR code is shown in Listing 4.11.

*Listing 4.11: Array Access Example IR After SafecodeVarMap*

```
1       ...
2       %arrayidx = getelementptr inbounds [10 x i32]* %a, i32 0, i64 %
            idxprom, !dbg !32, !acsl_safecode !30
3       store i32 %i.0, i32* %arrayidx, align 4, !dbg !32, !acsl_safecode
            !30
4       ...
5       !30 = metadata !{metadata !"assert␣i.0␣>=␣0␣&&␣i.0␣<␣10\0A"}
```

In Listing 4.12 we show an example of how, by attaching these metadatas to the single accesses, we are be able to instrument with our additional information also statement with multiple array accesses in a single line of code.

*Listing 4.12: Multiple Array Access Example*

```
1       ...
2       char * a1 = "@assert␣i>=0␣&&␣i<=10";
3       char * a2 = "@assert␣j>=0␣&&␣j<=5";
4       array[i] = array[i]+array2[j];
5       ...
```

*Listing 4.13: Multiple Array Access Example IR*

```
...
%array = alloca [10 x i32], align 16
%array2 = alloca [5 x i32], align 16
%idxprom = sext i32 %i to i64, !dbg !34
%arrayidx = getelementptr inbounds [10 x i32]* %array, i32 0, i64
    %idxprom, !dbg !34, !acsl_safecode !35
%0 = load i32* %arrayidx, align 4, !dbg !34, !acsl_safecode !35
%idxprom1 = sext i32 %j to i64, !dbg !34
%arrayidx2 = getelementptr inbounds [5 x i32]* %array2, i32 0, i64
     %idxprom1, !dbg !34, !acsl_safecode !36
%1 = load i32* %arrayidx2, align 4, !dbg !34, !acsl_safecode !36
%add = add nsw i32 %0, %1, !dbg !34
%idxprom3 = sext i32 %i to i64, !dbg !34
%arrayidx4 = getelementptr inbounds [10 x i32]* %array, i32 0, i64
     %idxprom3, !dbg !34, !acsl_safecode !35
store i32 %add, i32* %arrayidx4, align 4, !dbg !34, !acsl_safecode
     !35
...
!35 = metadata !{metadata !"assert i >= 0 && i <= 10"}
!36 = metadata !{metadata !"assert j >= 0 && j <= 5"}
```

### 4.5.2 Modifying SafeCode to use the annotations

Once *Aruna* has handled the task of propagating the annotation through the front-end and updating them to the correct IR registers, all we need is to make use of them has we wish in our application. SafeCode has two passes that can make use of them, therefore we augmented them so as to consider the external value analysis information coming from the annotations. The SafeCode *InsertGEPChecks* Pass and the *visitLoad* and *visitStore* methods of the *InstrumentMemoryAccesses* Pass are modified in a way that every time they are trying to insert checks for an out of bounds access they will test if there is a metadata containing a variable range or constant value. If the access using the GEP, LOAD or STORE instruction is safe (the operand is in the bounds of the array length) we can avoid to insert the check without loosing security margin.

## 4.6 Backend Optimizations

In this section we are analyzing existing LLVM optimizations and analysis and showing how to insert the additional information coming from the existing annotations inside these Passes. These Passes runs after the *PromoteMemoryToRegister* Pass so we need our *ACSLVarMapAfterM2R* Pass to be run immediately after it and before the other Passes we modified. Another approach could also be writing new optimization Passes from scratch based on these annotations.
Here we focus on the LLVM *Simple Constant Propagation* Transformation Pass and the *Lazy Value Information* Analysis Pass.

### 4.6.1 Improving Simple Constant Propagation Transformation Pass

The *Simple Constant Propagation* Transformation Pass implements constant propagation and merging. It searches for instructions involving only constant operands and replaces them with a constant value instead of an instruction. An example is showed in Listings 4.14 and 4.15.

*Listing 4.14: Before Simple Constant Propagation*

```
...
add i32 3, 4
...
```

*Listing 4.15: After Simple Constant Propagation*

```
...
i32 7
...
```

Since this pass could make definitions be dead the Dead Instruction Elimination is usually run after it.
This Pass runs on every function, it first inserts all the instructions in a work-list, then iterates on each of them and if their operands are constant it change them and propagates them replacing them in all their uses. A little code snippets is showed in Listing 4.16.

*Listing 4.16: Simple Constant Propagation Implementation*

```
    while (!WorkList.empty()) {
            Instruction *I = *WorkList.begin();
            WorkList.erase(WorkList.begin());    // Get an element from the worklist...

            if (!I->use_empty())                 // Don't muck with dead instructions...
                if (Constant *C = ConstantFoldInstruction(I, TD, TLI)) {
                    // Add all of the users of this instruction to the worklist, they might be constant propagatable now...
                    for (Value::use_iterator UI = I->use_begin(), UE = I->use_end();
                         UI != UE; ++UI)
                      WorkList.insert(cast<Instruction>(*UI));

                    // Replace all of the uses of a variable with uses of the constant.
                    I->replaceAllUsesWith(C);

                    // Remove the dead instruction.
                    WorkList.erase(I);
                    I->eraseFromParent();
                }
    }
```

Since we can have some annotations in our code about constant variables (such as *"assert x==7"* that the after the mapping could become

*"assert add == 7"*) we modified the implementation in order to improve the Pass optimization. The general annotation used is shown in Listing 4.17.

*Listing 4.17: Constant Annotation Used*

```
1       @assert value == const
```

After the *PromoteMemoryToRegister* and *ACSLVarMapAfterM2R* Pass every annotation now is about an identifier in SSA form. This means that we can propagate that value in the current basic block and all the others reachable from that basic block. We cannot simply propagate the SSA register in all its uses since the annotation only holds from that point and in the other basic blocks that come after in the CFG flow.
Therefore we modified the Pass in order to check not only if the instruction was "constant foldable" but also if there is an annotation asserting that that instruction has a constant value. As it happens in the original code the instructions where the value is propagated will be again inserted in the work-list to enable other cascading propagations and if the value gets propagated in all its users it will be removed. In addition our modified constant propagation also take into account the annotations about function arguments. If an argument is always used with a certain constant value also that value can be propagated.

*Listing 4.18: Constant Propagation Example*

```
1       #include <stdio.h>
2       int greaterThanZero(int x)
3       {
4           if (x > 0) return 1;
5           return 0;
6       }
7
8       int main()
9       {
10        int i = 0;
11        int j = 10;
12        int k = 0;
13        int z = 0;
14        while (i < j) {
15          k = greaterThanZero(j);
16          char * annotation = "@assert␣k␣==␣1";
17          z = k + 9;
18          if (k != 1) {
19            printf("this␣should␣not␣be␣printed");
20          }
21          else {
22            printf("k=%d",k);
23          }
24          i ++;
25        }
26        i += z;
27        return i;
28      }
```

In Listing 4.18 we give an example of a simple annotated program. The Frama-C plugin described in Section 4.2 is able to produce the annotation *"@assert k == 1"*. In Listing 4.19 we can see that the normal constant propagation pass LLVM is not able to propagate any value. Instead, as we can see in Listing 4.20, our modified version is able to propagate the value of *k* (named %call in the IR) thus enabling both the propagation of *z* (named %add in the IR) and the propagation of the *if* condition (which is translated in a compare instruction, named %cmp1 in the IR) which is always *false*. This will trigger the removal of the branch in a later Simplify CFG pass.

*Listing 4.19: Normal Constant Propagation Example*

```
...
while.cond:                                       ; preds = %if.end, %
    entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %if.end ]
  %z.0 = phi i32 [ 0, %entry ], [ %add, %if.end ]
  %cmp = icmp slt i32 %i.0, 10
  br i1 %cmp, label %while.body, label %while.end
while.body:                                       ; preds = %while.cond
  %call = call i32 @greaterThanZero(i32 10)
  %add = add nsw i32 %call, 9
  %cmp1 = icmp ne i32 %call, 1
  br i1 %cmp1, label %if.then, label %if.else

if.then:                                          ; preds = %while.body
  %call2 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
       ([27 x i8]* @.str1, i32 0, i32 0))
  br label %if.end

if.else:                                          ; preds = %while.body
  %call3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
       ([5 x i8]* @.str2, i32 0, i32 0), i32 %call)
  br label %if.end
...
```

*Listing 4.20: Modified Constant Propagation Example*

```
...
while.cond:                                       ; preds = %if.end, %
    entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %if.end ]
  %z.0 = phi i32 [ 0, %entry ], [ 10, %if.end ]
  %cmp = icmp slt i32 %i.0, 10, !dbg !27
  br i1 %cmp, label %while.body, label %while.end, !dbg !27

while.body:                                       ; preds = %while.cond
  br i1 false, label %if.then, label %if.else, !dbg !37

if.then:                                          ; preds = %while.body
  %call2 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
       ([27 x i8]* @.str1, i32 0, i32 0)), !dbg !38
  br label %if.end, !dbg !40

if.else:                                          ; preds = %while.body
  %call3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
       ([5 x i8]* @.str2, i32 0, i32 0), i32 1), !dbg !41
  br label %if.end


```

```
20
21      if.end:                                           ; preds = %if.else, %if.then
22        %inc = add nsw i32 %i.0, 1, !dbg !43
23        br label %while.cond, !dbg !44
24      ...
```

### 4.6.2 Improving Lazy Value Information Analysis Pass

The *Lazy Value Information* Analysis Pass is an interface for lazy computation of value constraint information. It is lazy so it will perform the analysis only when a dependent Pass will ask for some information about a value. The analysis is performed on a lattice structure where every *LVILatticeVal* type is showed in Listing 4.21.

*Listing 4.21: Lattice Information Type*

```
1       enum LatticeValueTy {
2           /// undefined - This Value has no known value yet.
3           undefined
4           /// constant - This Value has a specific constant value.
5           constant,
6           /// notconstant - This Value is known to not have the
                specified value.
7           notconstant,
8           /// constantrange - The Value falls within this range.
9           constantrange,
10          /// overdefined - This value is not known to be constant, and
                we know that it has a value.
11          overdefined
12      };
```

This lazy analysis is done by using a cache (*LazyValueInfoCache* Class) on which the value information is solved when needed. We store the information about constant and constantrange lattice values coming from the annotations in the cache together with the basic block in which they holds.

When the Pass will be asked the information about a value, the *solveBlockValue(Value *, BasicBlock *)* method that gets called is modified to search if there is an annotation about that value and uses it to improve the analysis.

The modified pass uses annotations of the kind shown in Listing 4.22.

*Listing 4.22: Value Info Annotation Supported*

```
1       @assert value == const
2       @assert value >= const1 && val <= const2
3       @assert value == const1 || ... || val == constN
```

The choice to improve this kind of analysis is motivated by the most frequent pass run during the BIND compilation with LLVM. As we can see in Listing 4.23 the Jump Threading and the Correlated Value Propaga-

tion are among the most recurrent passes used and they both depend on the Lazy Value Information Analysis.

*Listing 4.23: Most Frequent Passes during BIND compilation*

```
31975 ***  Simplify the CFG ***
31975 ***  Combine redundant instructions ***
16728 ***  Remove unused exception handling info ***
16728 ***  Promote by reference arguments to scalars ***
16728 ***  Function Integration/Inlining ***
16728 ***  Deduce function attributes ***
14400 ***  Canonicalize natural loops ***
12843 ***  Loop-Closed SSA Form Pass ***
12790 ***  'Correlated Value Propagation' ***
12790 ***  SROA ***
12790 ***  'Jump Threading' ***
12790 ***  Early CSE ***
8258 ***  Tail Duplication ***
...
```

### 4.6.3 Cascading Effects in other Transformation Passes

The advantage of modifying Analysis Passes is that then every Transformation Pass that depends on it can take advantage of the analysis improvements in order to perform better optimizations. Here we give an overview of the two Transformation Passes that depends on the *Lazy Value Information* Analysis Pass:

- The first one is the *Correlated Value Propagation* Transformation Pass. This pass handles the propagation of $\phi$s, selects, memory access targets, it simplifies compare instructions and switch cases that never fires. It uses the results from the *Lazy Value Information* Pass in order to test constant values.

  In Listing 4.24 we show an simple example where correlated value propagation while analyzing the operands of a Phi node is able to propagate constant values inside these operands by means of the modified Lazy Value Info analysis. The normal one will simply left the IR code as it is in Listing 4.25. As we can see in Listing 4.26 the values are propagated inside *y.addr.0*.

*Listing 4.24: Correlated Value Propagation Example*

```
...
char * annotation1 = "@assert x == 10";
if( x > y ){
        y = x;
        char * annotation2 = "@assert y == 10";
} else {
        y = x+x;
        char * annotation3 = "@assert y == 20";
}
z = y + x;
...
```

*Listing 4.25: Correlated Value Propagation Example IR*

```
...
%cmp = icmp sgt i32 %x, %y, !dbg !28
br i1 %cmp, label %if.then, label %if.else, !dbg !28

if.then:                                          ; preds =
    %entry
  br label %if.end, !dbg !35

if.else:                                          ; preds =
    %entry
  %add = add nsw i32 %x, %x, !dbg !36
  br label %if.end

if.end:                                           ; preds =
    %if.else, %if.then
  %y.addr.0 = phi i32 [ %x, %if.then ], [ %add, %if.else ]
  %add1 = add nsw i32 %y.addr.0, %x, !dbg !42
...
```

*Listing 4.26: Modified Correlated Value Propagation Example*

```
...
%cmp = icmp sgt i32 %x, %y, !dbg !27
  br i1 %cmp, label %if.then, label %if.else, !dbg !27

if.then:                                          ; preds =
    %entry
  br label %if.end, !dbg !34

if.else:                                          ; preds =
    %entry
  %add = add nsw i32 %x, %x, !dbg !35
  br label %if.end

if.end:                                           ; preds =
    %if.else, %if.then
  %y.addr.0 = phi i32 [ 10, %if.then ], [ 20, %if.else ]
  %add1 = add nsw i32 %y.addr.0, %x, !dbg !40
```

- The second one is the *Jump Threading* Transformation Pass. This Pass analyzes blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always jump to one of the successors, it forwards the edge from the predecessor to the successor by duplicating the contents of the block. A trivial example is showed in Listing 4.27. Here

the unconditional branch at the end of the first if can be forwarded to the else side of the second if.

*Listing 4.27: Jump Threading Example*

```
...
if (...) {
        ...
        x = 0;
        ...
}
if (x > 0) {
        ...
} else {
        ...
}
...
```

In addition if a block terminator (the last block instruction) is branching on a constant, it can simplify the terminator to an unconditional branch (this can occur due to threading in other blocks). This Pass uses the analysis to see if it can simplify branches and if there are value that are known by the *Lazy Value Information* Pass to be a constant in a predecessor, it uses that information to try to thread the current block.

In Listing 4.28 we show a trivial C program example in which simple annotations can help the compiler during the Jump Threading optimization.

*Listing 4.28: Detailed Jump Threading Example*

```
int foo(int x){
        if(x < 10){
                if(x > 8) {
                        char * annotation1 ="@assert_x==9";
                        x++;
                        char * annotation2 ="@assert_x==10";
                }
        }
        if(x == 10){
                return 0;
        }
        return 1;
}
```

Figure 4.2 shows how the code is translated by the compiler in the IR after the memory to register promotion. Since the memory to register promotion pass does not modify the resulting CFG, the source code program structure is still preserved and easy to understand from the graph showed below.

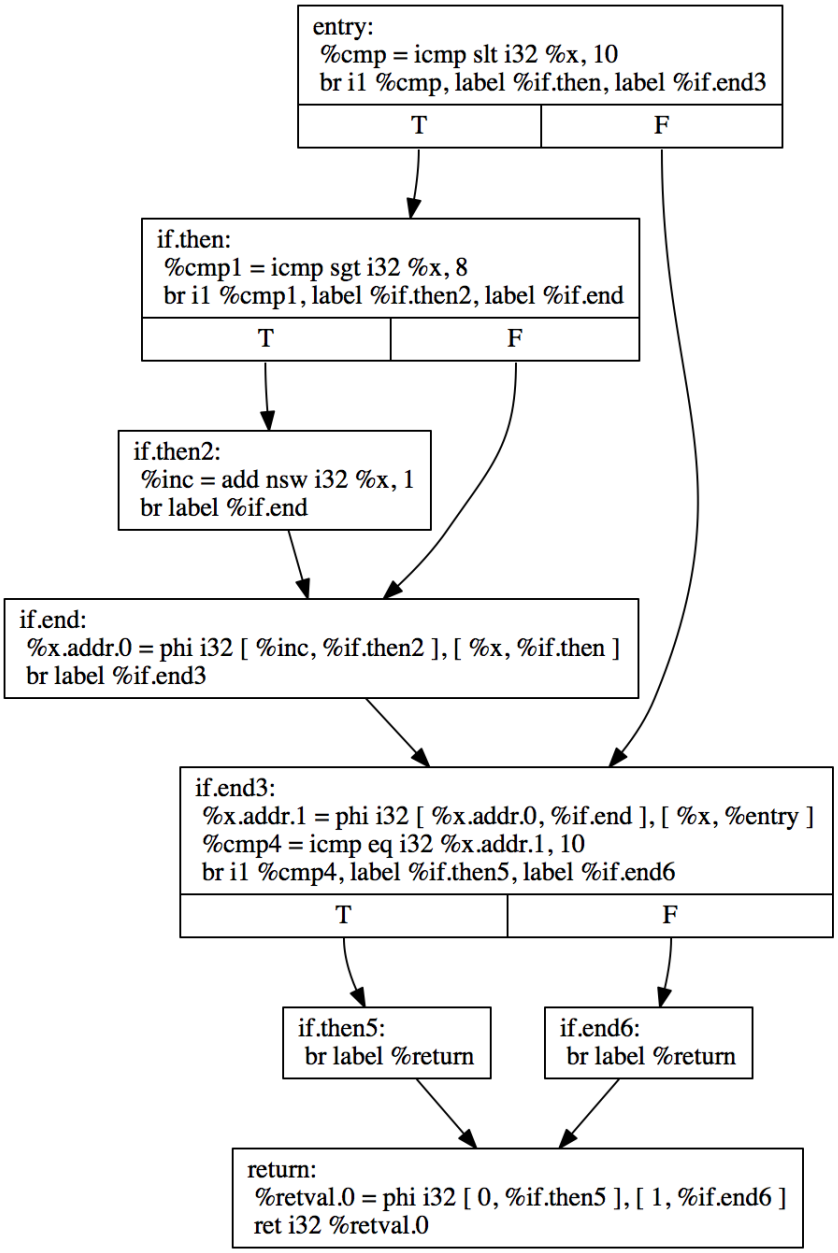


*Figure 4.2: CFG Before Jump Threading*

Figure 4.3 shows how the normal Jump Threading pass is able to thread a jump. It optimize the resulting code by removing two blocks (namely *if:end* and trivially *if:then5*) and making *if.then* block jump to *inf:end6* if the branch condition is false.

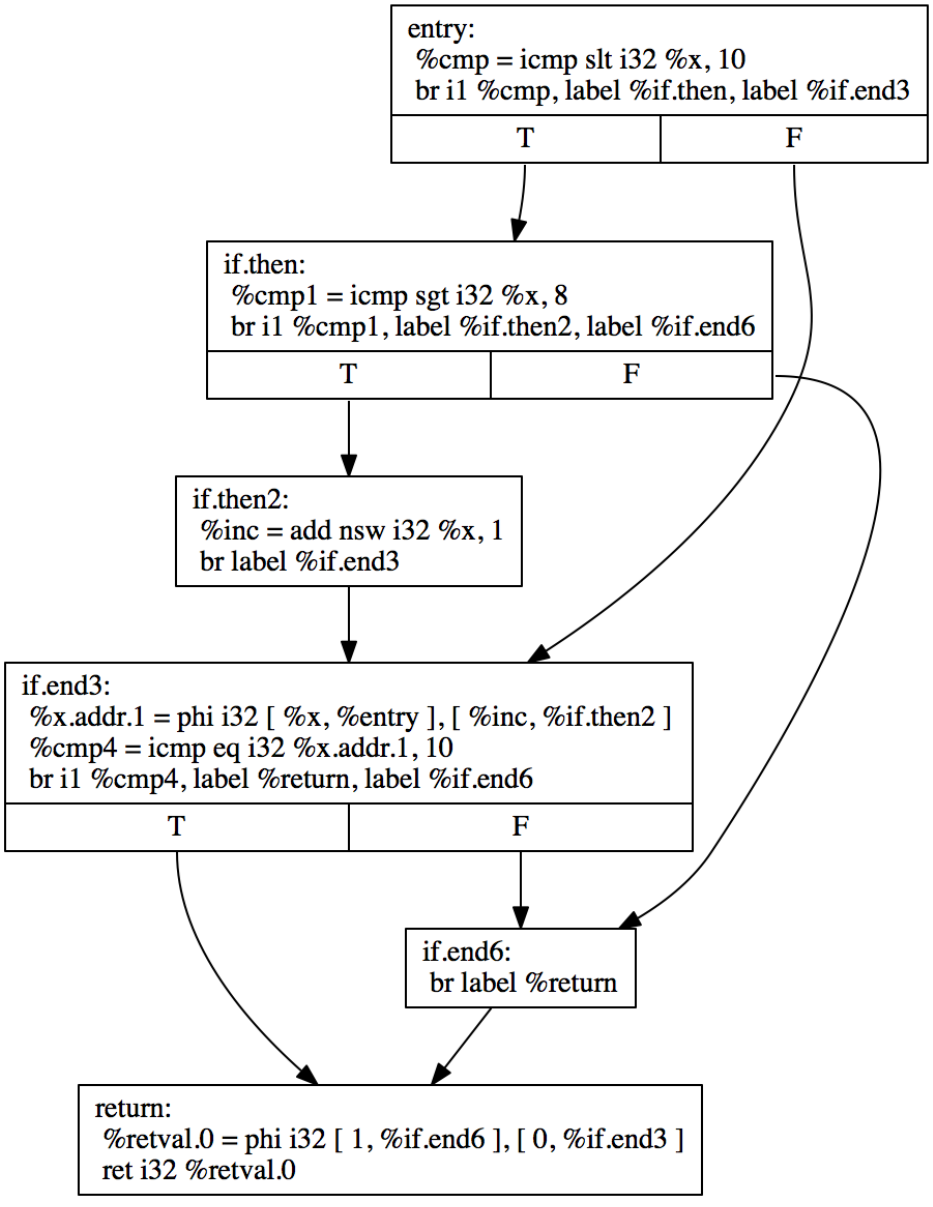


*Figure 4.3: CFG After Jump Threading*

Figure 4.4 shows how our modified Jump Threading Pass is able to thread an additional jump resulting in more optimized code compared to the normal one. As we can see in the picture below, the block *if.then2* gets removed and the jump from *if.then* is threaded to the *return* block.

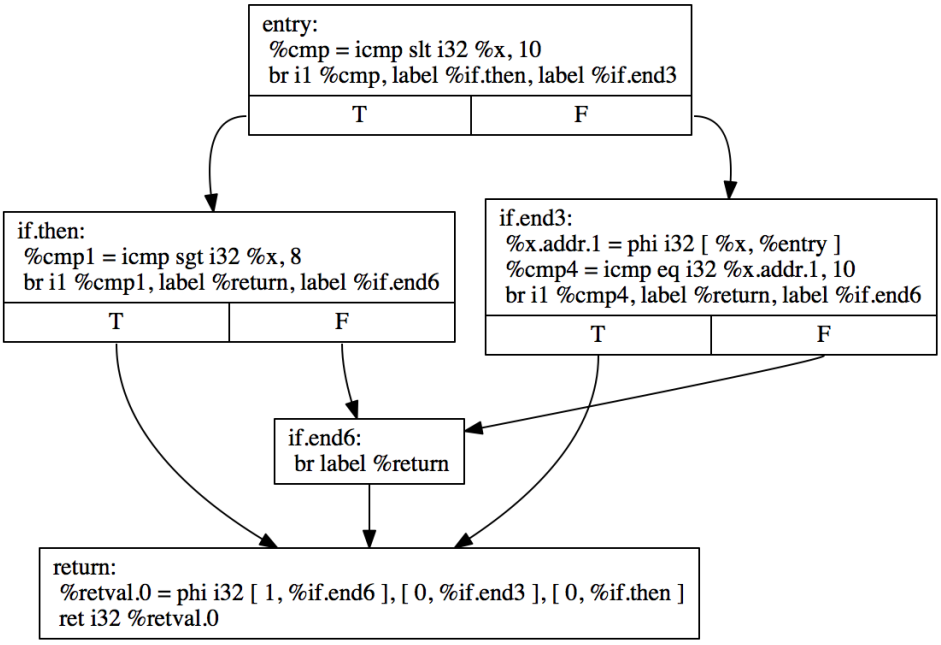


*Figure 4.4: CFG After Modified Jump Threading*

# Chapter 5

# Evaluation

Our framework *Aruna* was tested on five annotated C benchmarks described in Section 4.1.2. These benchmarks were annotated automatically using Frama-C and a custom cil-based plugin that injects the external annotations in every benchmark source code. These annotations were propagated by *Aruna* during the front-end phases and taken into account in the modified back-end passes.
To evaluate the uses of *Aruna* we augmented some back-end passes on the LLVM and SafeCode architecture and then tested the benchmarks compiled with the modified version against the normal ones. We both compared the statistics during the compilation and the runtime benefits in terms of code and time reduction. In addition for every optimization we show the results both when they are run immediately after memory to register promotion and in the proposed pipeline.
In this chapter follows a detailed comparison between the new results and the ones without the modifications. The results shows that the *Aruna* application in reducing the SafeCode checks have a nice impact both on the executable code size and execution time. In our experiments, the improvements over SafeCode are significant, in some cases up to two orders of magnitude. Instead, the impact on the modified optimizations side are not enough to motivate the use of the framework but there is still room for improvements as we will see in Chapter 6.

## 5.1 SafeCode Checks Reduction Results

Table 5.1 shows the number of checks the normal SafeCode is injecting in the code compared to the number of checks our modified version is inserting. As we can see we are able to a pretty high percentage of the

checks in almost all the benchmarks without loosing security margin.

*Table 5.1: SAFECODE CHECKS REDUCTION RESULT*

| SafeCode Checks Reduction Results | | | | |
|---|---|---|---|---|
| Benchmark | LOC | SafeCode Version | # Run-time Checks | % Run-time Checks Removed |
| CoreMark | 1831 | Normal | 309 | |
| | | Modified | 241 | 22.0065% |
| Susan | 1463 | Normal | 2251 | |
| | | Modified | 2008 | 10.7952% |
| MxM | 373 | Normal | 123 | |
| | | Modified | 102 | 17.0731% |
| Linpack | 579 | Normal | 318 | |
| | | Modified | 286 | 10.0630% |
| NECMatrix | 113 | Normal | 78 | |
| | | Modified | 32 | 58.9744% |

Table 5.2 show the impact of the check reduction in terms of code size. As expected our modified version is always generating smaller program since we are just avoiding the injection of additional checks.

*Table 5.2: SAFECODE EXECUTABLE SIZE REDUCTION RESULT*

| SafeCode Executable Size Reduction Results | | | |
|---|---|---|---|
| Benchmark | SafeCode Version | Code Size (bytes) | % Code Size Reduction |
| CoreMark | Normal | 1319475 | |
| | Modified | 1315271 | 0.3186% |
| Susan | Normal | 1375254 | |
| | Modified | 1366824 | 0.6130% |
| MxM | Normal | 977852 | |
| | Modified | 969324 | 0.8721% |
| Linpack | Normal | 1100586 | |
| | Modified | 1096384 | 0.3818% |
| NECMatrix | Normal | 792426 | |
| | Modified | 788276 | 0.5237% |

Table 5.3 shows the results in terms of execution time of the generated executable together with the lines of code of the benchmark.

The values in the Speed-Up column are computed as:

$$Speed - Up = \frac{NormalTime - ModifiedTime}{NormalTime}$$

As illustrated by Table 5.3, there is a wide variety in our runtime improvement results. This variety is due to several factors. First, not all checks are eliminated either because Frama-C is not able to produce assertions for every array access, or because it is not possible to determine the size of the arrays at compile time, or because our application does not support certain array accesses yet. 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, then the runtime improvement is not significant. If, however, they are located in a portion of the code that is executed often the improvements can be significantly better, without impairing the program's safety. The results were measured on an Ubuntu 12.04 machine, with Intel Xeon CPU @2.40GHz.

*Table 5.3: SAFECODE RUN-TIME REDUCTION RESULT*

| SafeCode Run-time Reduction Results | | |
|---|---|---|
| Benchmark | % Speed-Up | LOC |
| CoreMark | 0.3163% | 1831 |
| Susan | 3.4483% | 1463 |
| MxM | 12.0531% | 373 |
| Linpack | 95.9401% | 579 |
| NECMatrix | 46.6666% | 113 |

Specifically the poor results in the CoreMark benchmarks is due to the fact that we do not have annotations for the relevant portion of code, which is the one that is executed most often. The annotations that we have are about array accesses during initialization, which are executed only once.

Instead in the NEC-matrix benchmark, functions called only once from the main functions. At most double nested loops. No command line arguments required, therefore Frama-C finds correct information about all the index bounds. On the LLVM end, the size of the arrays is easy to determine since they are all created as global arrays of a fixed

known size

As NEC-matrix benchmark, Linpack Benchmark requires no command line arguments or user input, Frama-C gives useful information here if run with option -slevel 1000. The higher the slevel the more states Frama-C keeps in memory as it is going through the loops. In particular, the most useful information is that in a function (named *daxpy*) which is executed approx. 86% of the time according to the *Valgrind* [32] profiler. On the LLVM end, many functions receive the name of the arrays as pointer input parameters, therefore there is no easy way to get the size of the arrays during the optimization pass. We enumerated the call sites where the function is called and tried to see if the array being passed in input to it is allocated via a *malloc* or in the stack inside the caller function. In addition, in the source code, parameter values are computed inside the function call, e.g., *foo(a +j*x)*, where a contains the starting address of the array. So, if we have annotations about j and x, we can compute the exact value of the input parameter.

Both Susan and MxM are run with command line arguments, Frama-C value analysis needs to be given information about the values of those arguments. If this is done, then the annotations produced are somewhat good.

## 5.2 Back-end Optimizations Results

In this section we describe the results of our modified optimizations compared to normal one in LLVM. First we are going to show how every single optimization pass behaves on the input benchmarks, then we show the results of all the passes chained together in a pipeline and we present some additional information about the code size and execution time of the output executables.

### 5.2.1 Single Optimization Results

Table 5.4 shows that the external annotations have an impact in the propagation of constant values. These is due to the fact Frama-C Value Analysis is able to identify much more constant values that the normal Constant Propagation pass. Almost in every benchmark the constant information inside the annotation holds only in some blocks of the code (such as when the annotation is inside an if statement branch) therefore the information cannot be substituted in all the uses as in the normal constant propagation. That is why the most of the constant values in

the annotations are not killed. However the propagation of these values is able to have cascading effects in the propagation of other values in which these substitutions were done. In the NECMatrix benchmark Frama-C was not able to insert any information about constant values hence the modified optimization was not able to improve the results.

*Table 5.4: CONSTANT PROPAGATION RESULTS*

| Constant Propagation Results | | | | |
|---|---|---|---|---|
| Benchmark | Version | #Annot. Substituted | #Instr. Killed by Annot. | # Instr. Killed |
| CoreMark | Normal | | | 3 |
| | Modified | 18 | 0 | 73 |
| Susan | Normal | | | 1 |
| | Modified | 4 | 1 | 15 |
| Linpack | Normal | | | 0 |
| | Modified | 6 | 0 | 10 |
| NECMatrix | Normal | | | 0 |
| | Modified | 0 | 0 | 0 |

Table 5.5 shows how both in Susan and Linpack benchmarks the additional information in the annotated code is able to trigger the propagation of phi or comparison instructions. These results are poor, however they still confirm that these information can be effective in some real cases.

*Table 5.5: CORRELATED VALUE PROPAGATION RESULTS*

| Correlated Value Propagation Results | | | | | |
|---|---|---|---|---|---|
| Benchmark | Version | # Phi Prop. | # Select Prop. | # Cases Rem. | # Cmp Simpl. |
| CoreMark | Normal | 5 | 0 | 0 | 0 |
| | Modified | 5 | 0 | 0 | 0 |
| Susan | Normal | 0 | 0 | 0 | 1 |
| | Modified | 1 | 0 | 0 | 1 |
| Linpack | Normal | 0 | 0 | 0 | 0 |
| | Modified | 2 | 0 | 0 | 0 |
| NECMatrix | Normal | 0 | 0 | 0 | 0 |
| | Modified | 0 | 0 | 0 | 0 |

Table 5.6 shows that the annotations were not able to have an effect on trading jumps in any of the benchmarks. The modified version is still conservative and does not perform worse than the modified one. Hence,

at least there is nothing to loose in trying to use the information inside the annotations.

*Table 5.6: JUMP THREADING RESULTS*

| Jump Threading Results | | | |
|---|---|---|---|
| Benchmark | Version | # Jumps Threaded | # Terminators Folded |
| CoreMark | Normal | 12 | 1 |
| | Modified | 12 | 1 |
| Susan | Normal | 12 | 1 |
| | Modified | 12 | 1 |
| Linpack | Normal | 0 | 0 |
| | Modified | 0 | 0 |
| NECMatrix | Normal | 0 | 0 |
| | Modified | 0 | 0 |

### 5.2.2 Optimization Pipeline Results

Since the Constant Propagation pass is the first in our pipeline of passes the results its result in the pipeline are the same as the one reported in Table 5.4. However, the results in Table 5.7 shows that the instructions propagated in Susan and Linpack benchmarks in the previous Table 5.5 are already propagated during the constant propagation pass leaving the Correlated Value transformation unchanged.

*Table 5.7: PIPELINE CORRELATED VALUE PROPAGATION RESULTS*

| Pipeline Correlated Value Propagation Results | | | | | |
|---|---|---|---|---|---|
| Benchmark | Version | # Phi Prop. | # Select Prop. | # Cases Rem. | # Cmp Simpl. |
| CoreMark | Normal | 5 | 0 | 0 | 0 |
| | Modified | 5 | 0 | 0 | 0 |
| Susan | Normal | 0 | 0 | 0 | 0 |
| | Modified | 0 | 0 | 0 | 0 |
| Linpack | Normal | 0 | 0 | 0 | 0 |
| | Modified | 0 | 0 | 0 | 0 |
| NECMatrix | Normal | 0 | 0 | 0 | 0 |
| | Modified | 0 | 0 | 0 | 0 |

The results coming from the modified version of the jump thread-

ing are not better than the normal one as seen before in Table 5.6. In addition from these benchmarks the constant propagation in the first pipeline step seems not to enable better jump threading. The results in Table 5.8 shows that the results coming from the constant propagation pass have just a little impact in code size. The additional propagation of values and the instruction killed reduces the code size of the benchmarks. The last line of the table is left blank since we have no improvements on the NECMatrix benchmark. The Susan benchmark seems not to have any improvement, however the resulting IR is different and even if after the translation in machine code they have the same size the executable differ.

*Table 5.8: EXECUTABLE SIZE PIPELINE REDUCTION RESULTS*

| Executable Size Pipeline Reduction Results | | | |
|---|---|---|---|
| Benchmark | Version | Code Size (bytes) | % Code Size Reduction |
| CoreMark | Normal | 21,844 | |
| | Modified | 21,733 | 0.0505% |
| Susan | Normal | 37,313 | |
| | Modified | 37,313 | 0.0000% |
| Linpack | Normal | 16,209 | |
| | Modified | 16,214 | 0.0310% |
| NECMatrix | Normal | // | |
| | Modified | // | 0.0000% |

The performance results about Susan and Linpack benchmarks were taken over a 1000 iterations of the executables. Since the Unix *time* command seemed to be not reliable we slightly modified the benchmark logging the execution time at the beginning and the end of the main function and then averaging the result. To measure CoreMark List and CoreMark Matrix execution time we rely on the built-in Makefile that additionally logs the performance of the executables.The last line of the table is left blank since we have no improvements on the NECMatrix benchmark. However since the performance improvement is almost negligible in the benchmarks and the percentage of measurement error is higher than the performance speed-up we do not show the result table.

The effectiveness of the modified optimizations is correlated to the

information on the SCannotations. We inspected the resulting IR code and we have seen that the modified optimization are too few to make a real difference and that after all the other LLVM optimizations the resulting code is almost the same. In particular Function Inlining combined with Interprocedural Sparse Conditional Constant Propagation is already able to enable the same optimization that the most of the additional constant information inserted by the Frama-C Value Analysis plug-in enables.

# Chapter 6

# Future Work

Currently the implementation of the modified optimization passes depends on the order of their execution. This order is a constraint and it is due to the fact that the we have to guarantee that the informations in the annotations still hold after the transformation passes changes. The order may influence also later optimizations and it is not always guaranteed that the previous optimizations will not disable better later optimizations. Therefore one of the major drawbacks of our current approach is that we need a way to propagate the changes in annotations to remove this constraint.

In *Witnessing Program Transformations* [5] it is described how we can use witnesses both to validate the pass transformation and to correctly propagate invariants inside the code (for example our annotations). By implementing a transformation witness inside of the modified passes it will be possible to propagate the changes inside the annotations and to guarantee the correctness of their informations. This will enable a broader range of passes that could be modified in order to take advantage of the informations inside the annotations without worrying about the correctness of the changes in the annotations. These changes are automatically certified by the theorem prover by checking the witness, the input and the output program.

In addition to the new optimizations that can take into account the *Value Analysis* information it could be interesting to use other kind of analysis that can be helpful for different optimizations. Our framework design makes it easy to exploit different kind of analysis since it relies on a standard annotation language (and the a pass that map this information to the IR values) that act as an interface between the back-end and the different kind of analysis before the front-end.

Eventually merging the achievements from run-time checks removal, witness generation and optimization improvements we will be able to build a compiler able to both perform aggressive optimizations and defend code against security flaws.

# Chapter 7

# Conclusion

In this thesis we described the concept of program annotation, the different sources of annotated code and how nowadays compilers are ignoring this additional information. We described the approach we followed to build our framework *Aruna* , which allows the compiler to take advantage of the annotations to improve different aspects of the compilation process. Namely, our framework is able to reduce both the trade-off between security and execution time of a compiled program and to improve current compiler optimizations. In this work we inspected different ways to use additional information about variable range values so that the compilation can be more effective. We presented in detail how our approach can be useful in a state of the art compiler such as LLVM and how the framework was built in order to be highly reusable with different kind of program annotation sources. We evaluated the effectiveness of the approach on real world benchmarks using the information coming from the Frama-C Value Analysis plug-in. We showed how our framework is able to reduce the trade-off between security enforcement and performances, having impact both on the executable code size and the execution time by removing SafeCode checks for secure accesses. However, on the modified optimizations side, we still need to integrate the framework with the use of transformation witnesses. This will enable the annotation propagation during the back-end passes making them available not only at the beginning of the compilation process but through the whole LLVM passes pipeline. This will allow a broader use of *Aruna* inside additional LLVM optimizations so that the slight compilation time overhead introduced by our framework motivates the use of it.

# Cited Literature


[1] Schenk, Eric and Guittard, Claude: Crowdsourcing: What can be Outsourced to the Crowd, and Why?, Workshop on Open Source Innovation, Strasbourg, France, 2009.

[2] Lattner, Chris and Adve, Vikram: LLVM: A compilation framework for lifelong program analysis & transformation, Code Generation and Optimization, 2004.

[3] Cuoq, Pascal and Kirchner, Florent and Kosmatov, Nikolai and Prevosto, Virgile and Signoles, Julien and Yakobowski, Boris: Frama-C, Software Engineering and Formal Methods, 2012.

[4] Canet, Géraud and Cuoq, Pascal and Monate, Benjamin: A value analysis for C programs, Source Code Analysis and Manipulation, 2009.

[5] Namjoshi, Kedar S. and Zuck, Lenore D.: Witnessing Program Transformations, Static Analysis, 2013.

[6] SAFECode: Static Analysis For safe Execution of Code. http://safecode.cs.illinois.edu/index.html

[7] LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html

[8] Source Level Debugging with LLVM. http://llvm.org/docs/SourceLevelDebugging.html

[9] Crowd Source Formal Verification (CSFV). http://www.darpa.mil/Our_Work/I2O/Programs/Crowd_Sourced_Formal_Verification_(CSFV).aspx

[10] The LLVM Compiler Infrastructure. http://llvm.org

[11] LLVM Analysis and Transform Passes. http://llvm.org/docs/Passes.html

[12] Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25

[13] ANSI/ISO C Specification Language. http://frama-c.com/acsl.html

[14] Clang: a C language family frontend for LLVM. http://clang.llvm.org

[15] GCC Wiki. http://gcc.gnu.org/wiki/Speedup_areas

[16] Meyer, Bertrand: Eiffel: the language, 1992.

[17] Muchnick, Steven: Advanced compiler design and implementation, 1997.

[18] Burdy, Lilian and Cheon, Yoonsik and Cok, David R. and Ernst, Michael D. and Kiniry, Joseph R. and Leavens, Gary T. and Leino, K. Rustan M. and Poll, Erik: An overview of JML tools and applications, 2005. http://dx.doi.org/10.1007/s10009-004-0167-4

[19] Ron Cytron and Jeanne Ferrante and Barry K. Rosen and Mark N. Wegman and F. Kenneth Zadeck: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, 1991. http://doi.acm.org/10.1145/115372.115320

[20] Chris Lattner: LLVM: An Infrastructure for Multi-Stage Optimization, 2002.

[21] Necula, George C and McPeak, Scott and Rahul, Shree P and Weimer, Westley: CIL: Intermediate language and tools for analysis and transformation of C programs, Compiler Construction, 2002.

[22] GCC Function Specific Option Pragmas. http://gcc.gnu.org/onlinedocs/gcc/Function-Specific-Option-Pragmas.html#Function-Specific-Option-Pragmas

[23] GCC Loop-Specific Pragmas. http://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html#Loop-Specific-Pragmas

[24] Simone Pellegrini: An experimental framework for Pragma handling in Clang, 2013. http://llvm.org/devmtg/2013-04/pellegrini-slides.pdf

[25] Verigames. http://www.verigames.com

[26] Xylem, the <code> of plants. http://xylem.verigames.com

[27] F. Ivani, Z. Yang, M.K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform, Computer Aided Verification, 2005. http://www.nec-labs.com/research/system/systems_SAV-website/fsoft-publications.php

[28] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles and Boris Yakobowski: Frama-c: a software analysis perspective, 2012.

[29] Dirk Beyer, ThomasA. Henzinger, Ranjit Jhala, and Rupak Majumdar: The software model checker BLAST, 2007.

[30] Necula, George C. and McPeak, Scott and Rahul, Shree Prakash and Weimer, Westley: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, 2002. http://dl.acm.org/citation.cfm?id=647478.727796

[31] Java Annotations. http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

[32] Valgrind, an instrumentation framework for building dynamic analysis tools. http://valgrind.org

# Appendix A

# ACSL Supported Grammar

<annotation> → <stmts> END

<stmts> → <stmt> | <stmts> <stmt>

<stmt> → TASSERT <expr> | TREQUIRES <expr> | TENSURES <expr>

<expr> → <ident> | <numeric> | TLPAREN <expr> TRPAREN |

| <expr> TCEQ <expr> | <expr> TCNE <expr> |

| <expr> TCLT <expr> | <expr> TCGT <expr> |

| <expr> TCLE <expr> | <expr> TCGE <expr> |

| <expr> TCANDAND <expr> | <expr> TCOROR <expr>

<ident> → TIDENTIFIER

<numeric> → TINTEGER | TDOUBLE | TMINUS TINTEGER |

| TMINUS TDOUBLE

# Appendix B

# Interval Algebraic Structure

In this chapter we describe the algebraic structure used to model the range computation for SSA variables. The underlying set, the contiguous interval of the possible value for an integer variable, is the set of integers (positive and negative) pairs *NXN*.

In Listing B.1 follow a series of range and constant information about integer variables modeled as intervals:

*Listing B.1: Interval Examples*

```
1       i<=0 && i<=1000          -> i in [0,1000]
2       j==-50                   -> j in [-50,-50]
3       k==0 || k==1 || k==2     -> k in [0,2]
```

On this carrier set we define the internal sum and multiplication operations as follow:

- *Interval Sum* (+):
  $\forall x, y, w, z \in N, [x, y] + [w, z] = [x + w, y + z]$
- *Interval Multiplication* (*):
  $\forall x, y, w, z \in N, [x, y] + [w, z] = [i, j]$
  where $i = min(x*w, x*z, y*w, y*z)$ and $j = max(x*w, x*z, y*w, y*z)$

Listing B.2 shows how these operations are useful to propagate range information during SSA assignments in a basic block.

*Listing B.2: Interval Operation Examples*

```
1       suppose holds:
2
3               %i<=0 && %i<=10          -> %i in [0,10]
4               %j==-5                   -> %j in [-5,-5]
5               %k<=-2 && %k<=3          -> %k in [-2,3]
6
7       basicblock instructions:
8
9               %add1 = add %i %j        -> %add1 in [0,10]+[-5,-5] =
                    [-5,5]
10              %mul1 = mul %k %i        -> %mul1 in [-2,3]*[0,10] =
                    [-20,30]
11              %mul2 = add 3 5          -> %mul2 in [3,3]*[5,5] = [15,15]
12              %sub1 = sub %add1 %mul2 -> %sub1 in
                    [-5,5]+([-1,-1]*[15,15]) = [-20,-10]
```