TU WIEN Informatics

Interprocedural Constant Loop Bound Propagation for Patmos Architecture

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Jérôme Hue,
Matrikelnummer 12123713

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Emad Jacob Maroun

Wien, 7. Februar 2024

Jérôme Hue	Peter Puschner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Interprocedural Constant Loop Bound Propagation for Patmos Architecture

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Jérôme Hue,
Registration Number 12123713

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Assistance: Emad Jacob Maroun

Vienna, February 7, 2024

Jérôme Hue	Peter Puschner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Jérôme Hue,

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschlieÿlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Februar 2024

Jérôme Hue

Danksagung

Ich möchte meinem Mitbetreuer, Herrn Emad Jacob Maroun, und meinem Betreuer, Herrn Peter Puschner, für die Zeit, die sie sich für mich genommen haben, und die wertvollen Ratschläge, die sie mir gegeben haben, danken. Auch meinen Eltern danke ich herzlich für ihre Unterstützung, insbesondere für das Korrekturlesen meines Berichts.

Acknowledgements

I would like to thank my supervisor, Mr. Emad Jacob Maroun, as well as my supervisor Mr. Peter Puschner for their time, guidance, and valuable advice. My sincere appreciation also goes to my parents for their support, especially in proofreading my report.

Kurzfassung

Echtzeitsysteme erfordern nicht nur genaue Berechnungen, sondern auch die strikte Einhaltung zeitlicher Beschränkungen. Der Patmos-Prozessor, der für Echtzeitanwendungen entwickelt wurde, unterstützt eine Technik zur Codegenerierung, die als Single-Path-Code bekannt ist. Diese Technik stellt sicher, dass alle Programmausführungen konsistent einem einzigen Pfad folgen, unabhängig von Laufzeitereignissen. Ein entscheidender Aspekt beim Erreichen dieser Vorhersagbarkeit ist die Verfügbarkeit von Schleifengrenzen während der Kompilierung. Die korrekte Generierung von Code hängt davon ab, dass die genaue Anzahl der Iterationen von Schleifen bekannt ist, die zur Kompilierungszeit Zeit bereitgestellt wird.

Um dies zu erleichtern, bietet der Patmos-Compiler Unterstützung für ein Pragma, das es Entwicklern ermöglicht, die unteren und oberen Grenzen von Schleifen direkt im C-Code anzugeben. Diese Informationen werden dann verwendet, um Single-Path-Code zu erzeugen, der garantiert, dass Schleifeniterationen genau wie definiert ablaufen. Eine Herausforderung ergibt sich jedoch, wenn die Schleifengrenzen von Funktionsargumenten abhängen. In solchen fällen werden Schleifenbegrenzungspragmas unwirksam, da sie nicht in der Lage sind nicht konstante Werte zu berücksichtigen, was zu Kompilierungsfehlern führt.

Diese Masterarbeit untersucht die Optimierung von Echtzeitsystemen mit dem Patmos-Prozessor und konzentriert sich auf die Herausforderung, dass Schleifengrenzen von Funktionsargumenten abhängen. Die Forschung untersucht alternative Strategien zur Umgehung dieser Einschränkung und zielt darauf ab, die Anwendbarkeit der Single-Path-Code-Generierung in Szenarien zu verbessern, in denen Schleifengrenzen dynamisch von Laufzeitparametern abhängen. Die Ergebnisse tragen dazu bei, die Möglichkeiten der Entwicklung von Echtzeitsystemen auf der Patmos-Plattform zu verbessern, indem sie es ermöglichen, Code zu schreiben, der vorher nicht verarbeitet werden konnte.

Abstract

Real-time systems demand not only accurate computation but also strict adherence to temporal constraints. The Patmos processor, designed for real-time applications, supports a code generation technique known as single-path code. This technique ensures that all program executions consistently follow a singular path, independent of runtime events. A crucial aspect of achieving this predictability is the availability of loop bounds during compilation. The correct generation of code relies on knowing the exact number of iterations loops will perform, provided at compile time.

To facilitate this, the Patmos compiler provides support for a pragma allowing developers to specify the lower and upper bounds of loops directly in C code. This information is then utilized to generate single-path code, guaranteeing loop iterations occur precisely as defined. However, a challenge arises when loop bounds are contingent upon function arguments. In such cases, loop bound pragmas are rendered ineffective due to their inability to accommodate non-constant values, resulting in compilation failures.

This master's thesis explores the optimization of real-time systems using the Patmos processor, with a specific focus on addressing the challenge of loop bounds depending on function arguments. The research investigates alternative strategies to circumvent this limitation, aiming to enhance the applicability of single-path code generation in scenarios where loop bounds dynamically depend on runtime parameters. The findings contribute to advancing the capabilities of real-time systems development on the Patmos platform, making it possible to write code that couldn't be handled before.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In today's interconnected world, real-time systems are ubiquitous, playing critical roles in various domains, including healthcare, aerospace, telecommunications, and industrial automation. Real-time systems, in contrast to conventional systems, face an additional challenge: the imperative to meet stringent temporal constraints[1]. In essence, these systems must not only deliver accurate results, but must do so within a specified time frame.

To ensure the adherence to these temporal constraints, techniques have been developed to analyze the Worst-Case Execution Time (WCET) of a program. WCET analysis is essential in real-time systems, as it provides an upper bound on the time it takes for a program to complete its execution under the most adverse conditions. Meeting WCET guarantees is crucial, as it ensures that the system remains predictable and reliable in scenarios where timing is a vital requirement. For instance, in safety-critical applications like avionics systems in planes, accurate timing guarantees are essential for tasks such as flight control and navigation. Any deviation from the expected execution time could lead to potentially catastrophic consequences.

Despite the importance of WCET analysis, its practical implementation remains a challenge [2][3], primarily due to the ever-increasing complexity of modern hardware and software architectures. As technological advancements continue to push the boundaries of performance and functionality, understanding and estimating the WCET of programs becomes an intricate task.

As a result, a solution has been proposed: the Single-Path Approach. The Single-Path Approach consists of ensuring that a program only has a single execution path [4][5], making WCET analysis simpler by having to consider only one execution path. Our work fits within this framework and specifically focuses on optimizing loops.

1.2 Aim of the work

In order to produce single-path code, the compiler backend needs to convert different control structures (*if-else* statements, function calls, and loops).

With loops, the transformation needs to know the maximum number of iterations the loop will execute during the program runtime. This fixed value is referred to as constant loop bound. It needs to be known at compile time, thus provided in the source code using a pragma. As of today, developers can only define constant loop bounds as fixed integers.

This project seeks to address this limitation by allowing the loop bound to be dependent on a variable given as an argument to be propagated across function calls boundaries. In some cases, this can lead to a performance gain for the loops that do not iterate their maximum number of times every time. Additionally, it simplifies programming as developers no longer need to manually calculate this maximum number, easing the burden of the programmer.

The expected outcome of this project would be a modification to the Patmos compiler that allows loop bounds dependent on function arguments to be declared using the loop bound pragma. This would enable the generation of single-path code in such cases, and therefore make development for real-time systems easier.

1.3 Contributions

The contributions of this master's thesis project are listed below:

- A general algorithm to propagate constants across function boundaries to be used as loop bounds,
- The implementation of our algorithm in the LLVM-based Patmos compiler.

1.4 Outline

In the next chapter, we will provide comprehensive background information essential for the understanding of this project. This includes presentation of predicated instructions, an explanation of control-structure transformations made by the single-path code generator and basic definitions of control-flow graph (CFG), a representation of a program execution. We will also cover the single-path algorithm as an operation on a CFG as well as relevant details of its implementation. Lastly, we will cover related works of interest. In Chapter 4, we will detail the proposed approach in a theoretical manner, focusing on its conceptual difficulties and specificities. In Chapter 5, we will describe the implementation of our solution in the LLVM-based Patmos Compiler. This includes modifying both its frontend and backend, as well as a necessary work on some related parts of the compiler to accommodate for our modified loop handling. Next we discuss how we can validate and

benchmark our solution, using the existing built-in test framework, a de facto benchmark suite for WCET related work. We will also target some specific programs that are improved by our work, and demonstrate the possibilities it offers. Finally, Chapter 7 will conclude and provide some reflection on further work directions.

CHAPTER 2

Background

In this chapter, we offer background information essential for a thorough understanding of the work conducted. The emphasis is on presenting key technical aspects and contextual relevant details.

Section 2.1 discusses predicated instructions and the Patmos architecture. Section 2.2 provides a review of basic definitions for formally representing a program. Section 2.3 introduces the concept of single-path transformation. Lastly, Section 2.4 covers the implementation of Single-Path Code.

2.1 Patmos and Predicated Instructions

The Patmos instruction-set architecture (ISA) – part of the T-CREST project [6] – was intentionally designed for real-time systems, emphasizing time predictability and optimizing for low WCET [7]. In Patmos, each instruction is associated with one of eight predicate registers to enable or disable the instruction[8]. The ISA is fully-predicated, meaning every instruction type requires a predicate. Instructions also feature a predicate negation flag, indicating whether the instruction is enabled when the predicate register is true or false. For instance, the `add (p1) r1 = r1, r2` instruction represents an addition operation predicated on the `p1` register, with the option to negate the predicate using an exclamation mark `(!p1)`. If the predicate evaluates to true, the instruction is executed as usual. However, if the predicate evaluates to false, the instruction is transformed into a no-operation (`NOP`) : its execution does not modify the state of the program, but still takes some time.

Predicated instructions are shown in the two code snippets below. They present an update of the variable x. In both versions, what variable x gets updated to depends on how *cond* is evaluated. In Listing 2.1, branches are employed for implementing conditional behavior, while Listing 2.2 opts for predicated instructions to obtain the same result.

Listing 2.1: Update of x by branching

```
cond := ...
if (!cond) goto Lelse
Lthen :
  x := a + 1
  goto Lend
Lelse :
  x := b - 2
Lend :
  ...
```

Listing 2.2: Update of x by predicated assignments

```
cond := ...
( cond) x := a + 1
(!cond) x := b - 2
```

2.2 Basic Definitions

Modern compilers work by translating source code (that means scanning, parsing, analysing it) into an intermediate representation (IR) in what is called the *frontend*. Subsequently, this IR undergoes optimization before being transformed into machine-specific code in the *backend*.

In the following subsection, we will describe what a Control-Flow Graph (CFG) is, and give some crucial basic definitions.

Basic Block : A basic block is a continuous code block with one entry point and one exit point.

Control-Flow Graph : A Control-Flow Graph is a directed graph where nodes correspond to basic blocks, and edges indicate transitions between these blocks (i.e. control flow path) [9]. *Example* : Figure 2.3 is an example of a CFG.

Furthermore, we can define relationship between nodes of those graphs :

A node b_i **dominates** a node b_k if b_i is on every path from the entry node of the graph to b_k. Similarly, a node b_i **post-dominates** a node b_k if b_i is on every path from b_k to all exit nodes [10]. *Example* : node b in 2.3 dominates every other node in the graph, except for a. Node g post-dominates node b.

A **Loop** in a CFG is a strongly connected component of the graph. Moreover, we defined a **Natural Loop** as having an entry node called the loop header that dominates every

other node in the loop [11]. *Example* : nodes $\{c, d, e\}$ are a loop in 2.3. Figure 2.4 illustrates the construction of a loop header graph, where the parent of each node is the loop header of the innermost loop the node is in.

Forward Control-Flow graph: A Forward Control-Flow Graph (FCFG) is the graph obtained by removing all loop edges from a CFG, with a loop edge being an edge leading to a loop header [11]. Such a graph is acyclic [12]. *Example:* Figure 2.6 is the FCFG obtained by removing the loop-edges of component $\{c, d, e\}$ from the CFG graph of figure 2.3.

Control-Dependence is defined as follows : Given two nodes b_i and b_j of a CFG G, b_j is control-dependent on b_i if and only if there exists a path from b_i to b_j with any node in this path post-dominated by b_j and b_i is not post-dominated by b_j [11]. Example: in figure 2.3 node f is control-dependent on b.

Lastly, a new relation, **constant-loop dominance** has been introduced in [13], where it is defined in the following manner : "A node x constant-loop dominates a node y (x *cldom* y) if every path from the entry to y visits x a fixed number of non-zero times". This relation is used to enable more optimizations. In single-path code, a function that is always called enabled is denominated as a "*pseudo-root* function" (because this is trivially the case for the root function, a function that is single-path, but is called from a non single-path function). If a function is called from a constant-loop dominant block (meaning the loop runs a fixed number of times), we can optimize it similarly to a root function. This type of function is called a "pseudo-root" function. Any function called from a root or pseudo-root in a constant-loop dominant block is also considered a pseudo-root. Pseudo-root are interesting because they can be optimized by removing their predicate argument, since they are always enabled. This in turn requires the calling instruction to a pseudo-root function to be predicated, so that the pseudo-root function is not called from a disabled path.

2.3 The Single-Path Transformation

Single-Path Transformation needs conversion of control-flow structures, and can be described as an algorithm transforming the CFG of a program.

2.3.1 Conversions

As described earlier, several techniques are used to transform code into single-path code, that we list here :

- *If-Conversion*: In this, the predicate assumes the value of the condition. The statements within the "if" block are associated with this predicate, and the statements within the "else" block are associated with the negation of this predicate.

- *Loop-Conversion*: For Single-Path code, it is needed to make every loop iterate the same number of times. To achieve this goal, a loop counter is introduced, initialized with the maximum number of iterations for the loop (this information is typically provided through an annotation in the source code of the program). Subsequently, the counter is decremented with each iteration, and the loop terminates when it reaches zero. Notably, not all loops iterate the maximum number of times they would run in traditional code on every execution, with the body of the loop predicated with the initial loop exit condition. This results in some instructions being 'wasted' when the number of iterations of the loop is less than its maximum number of iterations.

- *Procedure-Conversion*: Procedure-Conversion is the trickiest. It might seem intuitive to add a predicate before the function call instruction. However, if we do this, it is possible that we end up with a different number of function calls between two executions of our program, if a function is called in some paths but not in others. To mitigate this, functions are modified to accept an additional predicate parameter. This predicate is then used on the function's instructions. As a result, functions are always called in single-path code, but they can be called *enabled* or *disabled*. Finally, we must note that loop counters operations are not predicated, so that even if a function is called disabled, it contains the same number of instructions as a function that would be enabled.

As an example, figures 2.1 and 2.2 show the single-path transformation applied to the CFG of a function. Notice how colors in Figure 2.1 indicate conditions for branching. Block B is conditionally branching to C and D in traditional code, but in single-path code it leads to both. However, it is possible that C is predicated by a false predicate. Colors in Figure 2.1 indicate conditions for branching : for instance, only if the green condition is true will B lead to C. This transfers to Figure 2.2 : only if the green condition is true will instructions in C be enabled.

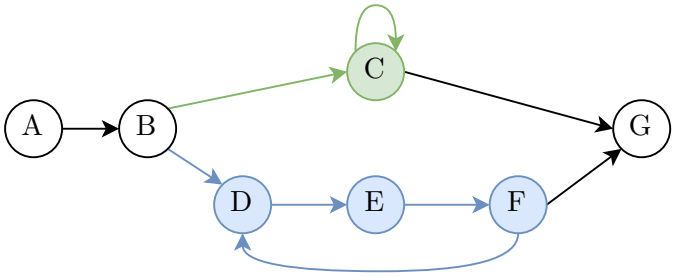

Figure 2.1: Traditional CFG

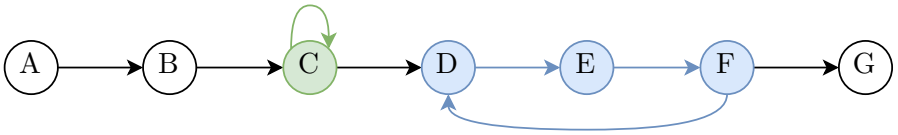

Figure 2.2: Single-Path CFG

2.3.2 Single-Path Code Generation Algorithm

Now that we have given the basic definitions, we can describe the transformation algorithm. This algorithm is described as operating on a CFG.

The different steps of the transformation algorithm, which is based on the RK-Algorithm by Park and Schlansker [14], are listed in [11] :

We start by identifying loops in the CFG. Subsequently, for each loop, its FCFG is constructed and partitioned based on control dependence. Predicates are then assigned to each class formed during this partitioning, with corresponding predicate definitions inserted at the source of control-dependence edges. Finally, the blocks are rearranged into a straight-line sequence in topological order. The single-path graph is composed from the rearranged blocks.

2.4 The Single-Path Code Generator

The single-path transformation is implemented using the LLVM Compiler framework. We will introduce LLVM, and describe how transformation is performed with it.

2.4.1 LLVM & Compilation Overview

LLVM (originally for *Low Level Virtual Machine*, but the name is no longer an initialism) is an open-source compiler framework written in C++ that started in 2000, developed by Christ Lattner at the University of Illinois [15]. It has gained a lot of popularity due to its modular and extensive design, as well as its emphasis on intermediate representation, providing a modern approach compared to GCC.

We are now going to break down the generic code-generation process of LLVM. More specifically, we will focus on the backend, that is generating assembly language from LLVM intermediate representation (IR).

In the LLVM code-generation process, the IR is represented in Static Single Assignment (SSA) form. This form ensures that each variable is assigned only once, which simplifies and accelerates various compiler optimizations.

One of the key features of SSA form is the use of PHI nodes. PHI nodes are a special kind of instruction that select a value based on the control flow of the program. They are necessary when a variable can be assigned a different value depending on the path of control flow.

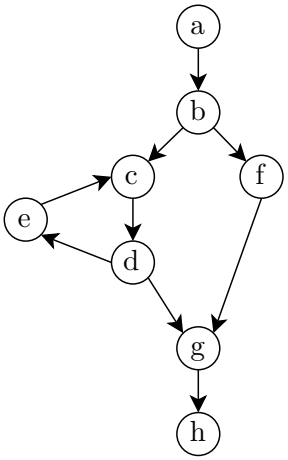

Figure 2.3: Example CFG

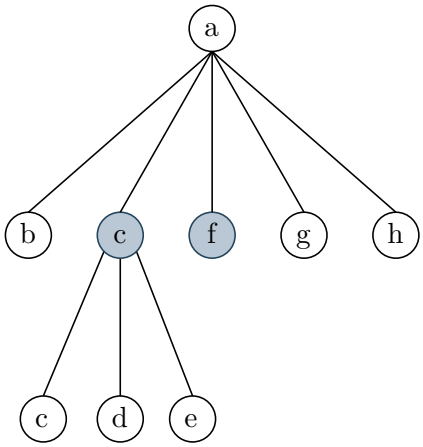

Figure 2.4: Loop Header tree of the graph

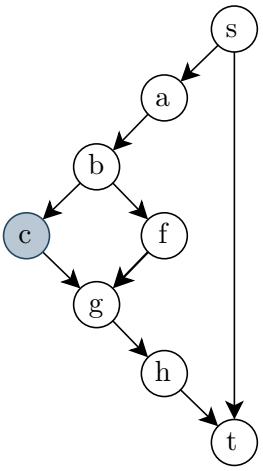

Figure 2.5: FCFG of the top-level pseudo-loop

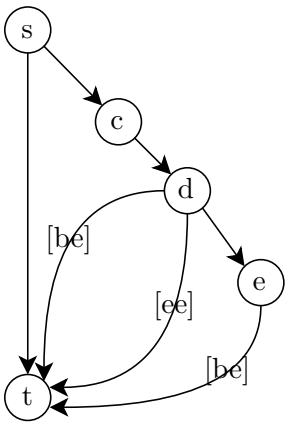

Figure 2.6: FCFG of the loop with header c

The code generation process of LLVM is constituted of seven steps listed and described below :

Firstly, the **Instruction Selection** phase translates the LLVM code into a directed acyclic graph (DAG) [16] of target instructions, utilizing virtual and physical registers (In order to allocate registers, compilers first assign values to virtual registers, of which there is an infinite amount. Later on during the compilation process, these virtual registers are substituted with physical registers.). Subsequently, the **Scheduling and Formation** phase orders the instructions and emits machine code. The optional **SSA-based Machine Code Optimizations** stage applies optimizations to the SSA-form produced by

the instruction selector. Following this, the **Register Allocation** phase transforms the target code from an infinite virtual register file to the actual register file of the target, introducing spill code and eliminating virtual register references. The **Prologue/Epilogue Code Insertion** stage inserts code for function prologue and epilogue, addressing stack space requirements. The **Late Machine Code Optimizations** phase handles final machine code optimizations, and finally, the **Code Emission** stage produces the target assembler format or machine code for the current function.

2.4.2 Single-Path Code implementation

The single-path transformation is implemented as a set of additional passes in the LLVM backend, modifying the normal flow of code generation described in 2.4.1. We won't describe each pass in detail, but will focus on the overall process and passes that really matter to us. The description of the implementation given in this subsection is based on unpublished work by the authors of [13].

This process can be described as revolving around two register allocations. The first one to allocate the general purpose registers, the second one to allocate the predicate registers. This code generation can be described in a few phases, that are introduced here, with relevant details explained in the next paragraph. In a first phase, code is prepared for Single-Path. Then, LLVM's standard register allocation is run. Next, we have a "cleanup" phase to prepare for the second register allocation and the main transformation, and finally the second register allocation is run.

The goal of the first phase is to prepare the code and make sure it meets the requirements of later passes. During this phase, each function is cloned two times, one version for single-path and one version for pseudo_root [1]. Later on, in that same phase, when would have determined which functions are to be single-path functions, and which are pseudo-root, the calls to such function will be rewritten by calls to one of their clone.

The cleanup phase consists of two steps. First the predicate registers that have been previously allocated are virtualised. The second one makes sure that disabled functions, whose loop counter is still being decremented, do not mess with the registers.

After cleanup, the main transformation takes place. Then, the LLVM infrastructure is used to run a second round of register allocation. Since only predicate registers are using virtual registers, this is in fact a predicate register allocation. The last thing that is done by single-path code is instruction scheduling.

[1] When compiling for single-path, one has to specify the root function of single-path. That means it is possible to have some functions that are not single path.

2.5 Related Work

We will cover related work in four different areas : predicated instructions, loop bound analysis, Constant-Execution Time and the LLVM framework.

The introduction of predicated instructions in the compilation process adds considerable complexity. This complexity requires updates in algorithms, analyses, and optimizations to accommodate potentially disabled instructions. Notably, traditional intermediate representations like LLVM typically do not directly incorporate predicated instructions. However, some research has explored modifications to static single assignment (SSA), a foundational concept in intermediate representations like LLVM, to include the modeling of predication [17].

Our work has to do with loop bounds. Proving a program termination is known to be undecidable [18], however for some practical cases, it remains possible to study loops and compute their bounds. An example of a tool doing this is LOOPUS, introduced in 2011 [19] and relying on LLVM, that can be used to compute loop bounds for C programs [20].

Constant execution time has now become the focus of single-path code: in [21], the authors examine automatic compilations techniques to achieve constant execution time programs. Apart from their application in real-time systems, another key domain for this technique is cryptography. In *timing attacks*, introduced by Paul Kocher in 2001, adversaries exploit variations in the execution time of cryptographic functions that operate on secret keys or information to extract confidential data [22]. This work has led to constant-time implementation of some cryptographic functions as a mitigation technique [23].

Speaking of LLVM, it continues to be widely used and be the subject of many published papers each year. Among the most prominent ones, we can distinguish of focus on bugs and formal verification, with the introduction of a bounded translation validation tool in 2021 [24].

CHAPTER 3

Program Analysis

3.1 Data-flow Analysis background

Given that we will employ analysis in our work, we will recall the theoretical foundation of data-flow analysis in this section.

Data-flow Analysis is a framework for providing properties about programs [25][26]. As its name might suggest, data-flow analysis can determine properties for each point of a program represented as a CFG.

In order to perform such an analysis, we first have to define transfer functions and control-flow constraints. A transfer function f_s connects the values available before (written $in[s]$) and after ($out[s]$) the execution of a statement s. In a data-flow analysis, information can be propagated *forward* and *backward*. For a *forward* analysis, we have $out[s] = f_s(in[s])$. For a *backward* analysis, $in[s] = f_s(out[s])$. To save time, we recognize that the information going in and out of the basic blocks ($out[b]$ and $in[b]$) are a single composition of the transfer functions for block statements, which allows us to use f_b as the transfer function for basic blocks.

Once the transfer functions are defined, is it necessary to state control-flow constraints. For this, the meet operator $\cap$ combines data-flow values from multiple branches: In a forward data-flow problem, in for a block b is computed from out of its predecessors:

$$in[b] = \bigcap_{s \in preds(b)} out[s]$$

In backward data-flow problems, out for b is computed from in of the successors:

$$out[b] = \bigcap_{s \in succs(b)} in[s]$$

The goal of a data-flow problem is to find a solution (that is, the values of *in* and *out* for all basic blocks in the flow graph) that satisfies these constraints. This can be done by a generic algorithm, given a specification as a *data-flow framework* $\langle D, \wedge, S, F \rangle$, where $D \in forward, backward$ is the direction of the analysis, $\wedge$ is the meet operator, S is a semilattice including a domain of definition V (see definition later) and $F : V \to V$ a family of transfer functions.

A lattice is a partially ordered set of elements (that is a set P and an operator $\leq$ such that $\leq$ is reflexive, antisymmetric, and transitive),

A semilattice can be define as an algebraic structure $(V, \wedge)$ composed of a set S and a operator meet $\wedge$, such for all $a, b, c \in V$, the operator is:

$$\begin{aligned} \text{Idempotent} : &\quad a \wedge a = a \\ \text{Associative} : &\quad a \wedge b = b \wedge a \\ \text{Commutative} : &\quad a \wedge (b \wedge c) = (a \wedge b) \wedge c \end{aligned}$$

In addition, a semilattice has a top element $\top$, such that for all $x \in V, x \wedge \top = \top$. A semilattice can also contain a bottom element $\bot$, such that for all $x \in V, x \wedge \bot = \bot$.

Finally, the meet element define an order on the domain. For all $x, y \in V$:

$$x \leq y \text{ if and only if } x \wedge y = x$$

Once all of this is defined, a data-flow framework can be solved using the following iterative algorithm shown in Algorithm 3.1. This algorithm starts by initialising every $out[b]$ to $\top$, then iterates until convergence, applying data-flow equation at each iteration.

Algorithm 3.1: Iterative algorithm for forward data-flow problems [25]

```
1  foreach basic block B do
2  |  OUT[B] ← ⊤;
3  end
4  while any out changes do
5  |  foreach basic block b ≠ ENTRY do
6  |  |  in[B] ← ∅;
7  |  |  foreach predecessor p ∈ preds(B) do
8  |  |  |  in[b] ← in[B] ∩ OUT[p];
9  |  |  end
10 |  |  out[B] ← f_B(in[B]);
11 |  end
12 end
```

3.1.1 Reaching Definitions

To determine, at the beginning of our loop, if the variable specified as our loop bound is correctly defined, we perform a standard *reaching definition* analysis. We now specify the data-flow framework for it.

A definition d of some variable v reaches point p of our program if and only if p reads the value of v and there exists a path from d to i that does not define v [27].

We will now specify the data-flow analysis tuple :

- The domain of our lattice is the set of definitions. The $\perp$ element will be U, the set of all definitions, and the $\top$ element will be $\emptyset$.
- Direction : *forward*
- Transfer function : $gen[b] \cup (x - kill[B])$,
  where $gen[b]$ is the set of definitions 'generated' (a new value is assigned to a variable) by block b and $kill[b]$ is the set of definitions killed (a definition is killed if at some point a variable is assigned a new value that overrides any previous definitions) by block b (and x is the argument of our transfer function, also a set of definitions).
- The meet operator is the set union operator $\cup$.

This algorithm terminates, because at each iteration, the $out[b]$ values of each basic block can only decrease, and lattice is of finite height.

More formally, to prove that the iterative algorithm for the data-flow framework terminates, we have to prove that it is monotonic, or to prove a stricter condition, such has distributivity : $f(x \wedge y) \leq x \wedge y$

In the case of reaching definitions, we have [25] :

$$GEN \cup (x \wedge y - KILL) = GEN \cup (x - KILL) \bigcup GEN \cup (y - KILL)$$
$$(x \wedge y - KILL) = (x - KILL) \bigcup (y - KILL)$$
$$(x \cup y - KILL) = (x - KILL) \bigcup (y - KILL)$$

3.2 Constant Propagation

Constant Propagation is a data-flow analysis whose goal is to determine expressions that evaluate to the same constant every time, and to replace these expressions by this constant[28]. In the general case, constant propagation is known to be undecidable, but it can be solved for some specific instances (simple constant, linear constant, conditional constant, ...).

Constant propagation is a data-flow problem. To solve it, the lattice used is a three levels lattice (or flat lattice). This lattice has a bottom element, a top element, and every possible constant value in the middle level.

An algorithm for constant propagation is the *Sparse Conditional Constant Propagation* (SCCP), introduced in 1991[29], and used in the LLVM compiler. This algorithm builds upon three other algorithms.

- Simple Constants [26].
- Sparse Simple Constant, an algorithm using the SSA form of programs [30].
- Conditional Constant, an algorithm that can find all constants that can be found by evaluating conditional branches having constant operands [31].

Sparse Simple Constant is faster than *Simple Constant*, but *Conditional Constant* can find more constants than both of these algorithms. *Sparse Conditional Constant* combines the best of both worlds, finding as many constants as *Conditional Constant* while being faster.

We will provide a short description of the SCCP algorithm, based on [27] and [29].

This algorithm starts by assigning a lattice value for each result of operation in the SSA form of the program. It then proceeds by iterating over two worklists until both of them are empty : a *Flow Worklist*, that contains control-flow edges, and *SSA Worklist*, that consists of SSA edges (here, SSA is represented as a graph). Additionally, it associates a flag `ExecuteFlag` to every flow graph edge.

During the initialization phase, SCCP assigns each lattice value to $\top$ and set each `ExecutableFlag` is false. The *Flow Worklist* is initialized with the edges that are leaving the procedure entry node, while the *SSA Worklist* is initialized with the empty set.

Next, the algorithm picks an edge in any of the two worklists, and process it.

If the edge is part of the *Flow Worklist*, SCCP looks if it marked as executed. If so, nothing is done. If not, then the edge is marked as executed (the `ExecuteFlag` is set to true), and SCCP visits all PHI functions (using the $visit - \phi$ procedure described below) at the start of the destination block pointed by the edge. Next, if only one of the ExecutableFlags of incoming flow graph edges for dest is true (that means that the

destination is visited for the first time), then it evaluates expressions in the destination node (using a *visitExpression* procedure detailed below).

If the edge is an SSA edge, and the first destination instruction is a PHI function, then we use $visit - \phi$ on it. If it is an expression, then we look at the `ExecuteFlag` for each flow graph node reaching this edge. If any of them is `true`, then *visitExpression* is performed, otherwise nothing happens.

If this item is part of the SSA worklist, and the destination is a PHI function, we use *visit-ϕ*.

$visit - \phi$ works as follows : Each operand of the function is evaluated. Then the value of v is equal to the meet of values of v_i. The lattice cell has the same value as the end destination of the ssa edge.

Evaluating an Expression is done as such : We evaluate the expression based on values of operators. If the result is the same as the old one, nothing is done Otherwise, if the expression if part of an assignment, the algorithm adds all outgoing SSA edges to the *SSA Worklist*. If the expression controls a conditional branch, then if the new result for the expression is $\top$, add all outgoing flow edges to the *Flow Worklist*. If the value is a constant , only add the flow graph edge executed as the result of the branch to *Flow Worklist*.

To obtain an interprocedural analysis, it is needed to discover the initial set of constants, propagate those constants, and model the transmission of values through procedures. Finding the initial set of constant can be done with SCCP, propagating them can be achieved using an iterative algorithm. The transmission of values through procedures is done by modelling each parameter by a *jump function*.

CHAPTER 4

Design

This chapter will describe the overall design of our solution. Section 4.1 will describe what our changes to the frontend should look like to the end user. In Section 4.2 will describe what our transformation pass will look like.

4.1 User perspective

From a user perspective, things are very simple. The user should just be able to specify which variable is used as an upper loop bound with a pragma, and define a lower bound as an integer in the same pragma. During the transformation from source code to LLVM IR, the value of the specified loop bound variable is unknown to the compiler. Therefore, we can't provide any sanity check except for the existence of the specified variable in the program.

To illustrate this, a code snippet from the TACLe[32] benchmark collection is presented in Listing 4.1. This snippet presents a function containing a loop, and the pragma giving minimum and maximum loop bounds. Since the loop bounds are to be given as integers, it is not yet possible to write a single function taking the maximum loop bound as an argument, as is suggested by the name of the function, `cover_swi10()`. Indeed, Listing 4.2 shows the `cover_main()` function of the program, displaying the use of three functions differing only in their iteration count. With our changes, it should be feasible to write the code presented in Listing 4.3: a new pragma `varloopbound` is used, with a minimum number of iterations of 0, and a maximum number of iterations of n. The three different calls of Listing 4.2 can thus be replaced by a single call to the function `cover_swi_n`, reducing code duplication and making the program more scalable.

```
int cover_swi10( int c )
{
  int i;

  #pragma loopbound min 10 max 10
  for ( i = 0; i < 10; i++ ) {
    switch ( i ) {
      case 0:
      ...
}
```

Listing 4.1: Code snippet from file cover.c, using integers as loop bounds

```
void _Pragma( "entrypoint" ) cover_main( void )
{
  cover_cnt = cover_swi10( cover_cnt );

  cover_cnt = cover_swi50( cover_cnt );

  cover_cnt = cover_swi120( cover_cnt );
}
```

Listing 4.2: Main function from file cover.c, that is calling three different functions differing only by the number of iterations

```
int cover_swi_n( int c, int n )
{
  int i;

  #pragma varloopbound min 10 max n
  for ( i = 0; i < n; i++ ) {
    switch ( i ) {
      case 0:
        c++;
        break;
      case 1:
      ...
}
```

Listing 4.3: Code snippet from file cover.c, using a variable as loop bound

4.2 Overall design

Once we know what the loop bound variable is, we have to determine if it is constant through all execution paths, in order for it to be used as the loop bound. For this we will rely on LLVM's own optimizations, notably constant propagation, which is already implemented in the backend[33]. It is essential to note that currently, the compiler cannot

be used with an optimization level lower than `O2`[1]. With this optimization level, LLVM can already optimize function calls by replacing variables that are constants with their values without hindering single-path code production. Lastly, being able to determine if the variable provided by the programmer as a loop bound is constant is not enough, due to the way function calls are treated in single-path code. Indeed, it is possible that a function is called disabled (meaning its body is predicated with false predicate). However, this type of function still requires the loop counter to run (as explained in subsection 2.3.1). This is a problem, because if the call to the function is disabled, the arguments will not get passed onto the stack, and trying to use them in the loop counter will result in an error. Thus, we also have to make sure that the function is always called enabled, and this is explained later.

4.2.1 Intraprocedural Case

The first case happens when the variable used for loop bound is defined inside the function, more precisely when there is definition of the loop bound in a path from the function entry to the machine code pseudo instruction storing our variable. In this case, we have two things to do :

- Determine the definitions of our loop bound that is reaching the loop header.
- Determine is this definition yields a constant loop bound.

In order to satisfy the first point, we start by performing a reaching definition analysis. The result of this analysis will be, for each statement, the set of definitions that may reach this statement. For the second, point, we have all the definitions reaching every point in our program, we can thus trace back the initial definition of our loop bound variable, and determine if it will be constant or not. In case it is not (for instance, it could be dependent on the input given to the program, or defined in a branch), we report an error. Doing so is just a matter of case disjunction on our instructions, thus an implementation detail.

Listing 4.4 below demonstrates this first case, where the loop bound variable r is defined inside the function's body. It is not sure that it is constant with just this snippet, as its value depends on x.

[1]This is for instance enforced in the TACLe benchmark suite, where the scripts used to build the test use the `O2` arguments regardless of the configuration

```
1  int func1(int x, int a)
2  {
3    int r = x+57;
4    r += 4;
5
6    #pragma varloopbound min 0 max r
7    for(int i = 0 ; i < r; ++i) {
8      a +=2 ;
9    }
10
11   return a;
12 }
13
14 int main() {
15   return func(1,10);
16 }
```

Listing 4.4: Intraprocedural Case

4.2.2 Interprocedural Case 1

Now let us investigate the case where the loop bound variable is one of the argument of the function (where it is not redefined in the path from the function entry to the loop, as we would fall back to the first case). This may seem like an oddly specific case, but in practice, we have a lot of such functions.

In order to handle this case, we can rely on LLVM built-in constant propagation, that can propagate constants and replace them by their values, even in complex cases, where the argument is constant in the caller of the caller.

Assume for now that the function we are in is always called enabled (that means its body is not predicated). If the argument has already been optimized by LLVM for each function call, then we can simply use the function argument to initialize the loop counter of the function. In the event that the argument has not been optimized, that means the LLVM built-in optimization passes were not able to determine that the argument/loop bound was constant, thus we stop here and return an error.

An example of such a case is given in the next figure. The loop will iterate 20 times then 10 times, whereas if we had it specified as a normal loop bound pragma, we would have to give 20 as the maximum loop bound, thus our loop would have iterated 40 times in total, with 10 times its body disabled in the second run.

```
int func(int a, int b)
{
  #pragma varloopbound min 0 max b
  for(int i = 0 ; i < b; ++i)
    a +=2 ;

  return a;
}

int main() {
  int r;
  // Loopbound is the second argument of func()
  r += func(1, 20);
  r += func(5, 10);
  return r;
}
```

Listing 4.5: Interprocedural Case 1

4.2.3 Interprocedural Case 2

The last case is similar to the second one, except that this time, the function we are looking at is not always called enabled.

Then we cannot simply use the register holding the function argument that is our loop bound, because the function argument is not passed on the stack when a function is disabled. What we do here is try to find the maximum value of our loop bound across all calls to this function in the program, and use it as a loop bound, making it generate the same code as if a normal loopbound pragma was used. An example of such case is shown in Figure 4.6.

```
int main() {
  int r;
  if(cond) {
    r += func(1, 20)
  } else {
    r += func(5, 10);
  }
  return r;
}
```

Listing 4.6: Interprocedural Case 2

CHAPTER 5

Implementation in LLVM

Our solution will be implemented in the Patmos Compiler, based on the LLVM framework. In a first short section, we will cover the changes needed to specify loop bounds as variables. In a second section, we will describe the implementation of our solution proposal, presented in section 4.2. Lastly, we will describe how we can use our work to improve the compiler with a specific application.

5.1 Variable Loop Bounds

This section is divided into two subsections. Subsection 5.1.1 explores the changes made to the frontend, while 5.1.2 describes the changes made to the backend.

5.1.1 Frontend changes

To implement our pragma, we rely on existing resources, specifically those created by developers on LLVM (see [34], [35] and [36]). Obviously, the implementation of the already existing pragma *loopbound* to specify loop bounds as integer will also serve as a reference. The syntax of this existing pragma is as follows :

```
#pragma loopbound min VALUE max VALUE
```

Where `VALUE` is an integer argument.

This new pragma, called `varloopbound`, has the following syntax:

```
#pragma varloopbound min VALUE max EXPR
```

Specifying the minimum number of iterations as a variable has also been implemented for this pragma, but the only use we can make of it is checking that a function iterates a fixed number of times. For instance, if we specify `#pragma varloopbound min size max size`, then the function always iterates `size` times, and some specific optimizations can be enabled. Since it doesn't modify the core of our design, we won't mention it in the rest of this document.

It is necessary here to state how metadata is stored: during the generation of the Intermediate Representation, loop bounds are not attached to a loop statement in the normal way. Instead, a function call is inserted into the code, and the function used is `llvm.loop.bound`, which is created in the IR for the sole purpose of keeping the min and max loop bounds (one might think of it as some sort of vehicle), and is not performing any computation (in fact, we do not even provide a definition for it, only a declaration, with attributes to make sure it is not optimized away). This choice of using a function instead of using the standard metadata mechanism provided by LLVM (and described in [37]) had to be made to prevent LLVM from removing metadata when merging two basic blocks. For our modification, we will call a new function `llvm.loop.varbound`, also taking two arguments. Using a new function instead of the existing `llvm.loop.bound` function is debatable, but it makes it easier to implement the conversion of this function to machine instructions later on, and let us also clearly separate our work of what has already been done, which is useful during debugging.

During the generation of LLVM IR, we inspect the previously generated code to find the instruction that defines our variable. We use this variable to create our function call. This is the main difference with the existing function, which could simply take a constant integer argument. We also have to create the definition of this function and add it to the generated IR.

Implementing this pragma differs with the existing in the way we are parsing our argument. Since they are not integer values, we store them as a string and convert it to a special class in LLVM. Then the emission of LLVM IR is handled by a function called `EmitLoopBounds`.

Listing 5.1 features a snippet of C code where a function called `add_to` of which the body contains a for loop whose loop bound is the second argument of the function. The pragma is highlighted in magenta. Listing 5.2 shows a part of the LLVM IR generated by compiling this function, and highlighted in magenta is the call to `llvm.loop.varbound` produced by our pragma.

```
int add_to(int x, int add)
{
  #pragma varloopbound min 0 max add
  for(int i = 0; i < add; ++i) {
    x++;
  }
}
```

Listing 5.1: Initial C source code

```
...
for cond:
  %0 = load i32, i32* %i, align 4
  %1 = load i32, i32* %add.addr, align 4
  %cmp = icmp slt i32 %0, %1
  call void @llvm.loop.varbound(i32 0, i32 %add)
  br i1 %cmp, label %for.body, label %for.end
...
```

Listing 5.2: LLVM IR generated from initial C source code

5.1.2 Backend changes

A few changes must be made in the backend. First, we need to modify the Instruction Selection phase, that will generate a pseudo instruction holding our loop bounds from the IR. Next, we need to modify the code initializing our Loop Counter, so that it can accept a variable and not just an integer. Eventually, the function `getLoopBounds()` is used in various places to check that loop bounds exist or using them, for each of the place where it is used we also check that variable loop bounds exists, not just integer ones.

5.2 Main Pass

We will first describe the main pass that uses the data-flow analysis described in Section 3.1, and we'll cover other changes to the backend in Section 5.3. We are thus not covering things in the order they take place in the compiler, since some changes described in Section 5.3 are taking place before the main pass described in the current section in this document.

Our main pass takes place before the first register allocation. It also takes place before the loop counter creation and the rewriting of function calls described in subsection 2.4.2.

For each function, we look for calls to `llvm.loop.varbound`. Then, for each of these calls, we determine if the provided loop bound is part of the function arguments. If it is, we verify that this variable has indeed been replaced by a constant by LLVM during its own optimizations (otherwise, we report an error). If it is not (i.e. the loop bound is defined inside the function), then we apply the analysis described earlier to see if it is

defined by a value known at compile time. Finally, for functions that are not always called enabled, we look for the maximum loop bound across all function calls, and replace the call to `llvm.var.loopbound` by a call to `llvm.loop.bound` with the constant integer.

The last sentence implies that we have a way to determine what functions are always called enabled, and this is provided by the constant-loop dominator analysis, as described in Section 2.2. At this point in the transformation, we already have access to two cloned functions (suffixed by `_sp_` and by `_pseudo_sp`[1]) as they have been cloned earlier, and a latter pass use constant-loop dominance to replace the functions always called enabled by the `_pseudo_sp` clone, whereas 'normal' functions that should be converted to single-path are replaced by their `_sp_` clones.

Any data-flow analysis is greatly simplified in LLVM IR, since it is in SSA-form, meaning each variable has only one definition. Our implementation of the reaching definitions analysis is very classical and uses a worklist algorithm, a well-known optimization over the iterative algorithm.

To begin, we map each instruction in our function to a unique index, representing the order in which they appear. Since definitions are uniquely identified by instructions, these indices are a convenient way to represent reaching definition information. Next, we define a class to represent this information, that is just a set of integers. Edges are represented by a pair of integers. The class holding our information contains procedures to join information. Instructions indices are also used in the worklist.

Then, we need to implement the transfer functions for different categories of LLVM IR instructions. For instructions that define variables (return a value), we update the reaching definitions based on the index of the defining instruction. For instructions that do not return a value, treat them as if they don't define any variables.

5.3 Making use of our variable

Next we need to modify the code initializing the loop counter, and instead of the upper bound as an integer, we can give it the variable specified as a loop bound.

Finally, there are many places scattered across the code where the transformation is checking that the code has loop bounds correctly set and sometimes tries to use them (as an example, the assembly emitter currently emits the loop bounds as comments alongside the assembly), so we have to handle all of these cases by also providing a way to check that variable loop bounds are defined, and if the current code expects an integer, remove that usage.

[1] _sp will be the single-path function and pseudo_sp the pseudo root function, as explained in 2.4.2 and 2.2

5.4 Handling of LLVM intrinsics

Finally, we cover an application of our work in this last subsection : the handling of LLVM's `memcpy` intrinsic.

The C library function `void *memcpy(void *dest, const void *src, size_t n)` copies n characters from the object pointed by `src` to the object pointed by `dest`.

The LLVM compiler has its own intrinsic `memcpy` function. It does so in order to generate more efficient code, as the behavior of the memcpy function can be optimized by different architectures.

For Patmos, the compiler currently checks that `n` is an integer value, and if it is, replaces the `memcpy` intrinsic by a loop doing the copy from `src` to `dest`. With our changes, we can lift this restriction, and generate the code for `memcpy` even if `n` is a variable. Our main transformation pass will then have the duty of verifying that this loop bound, as every other, is constant.

This loop is composed of multiple statements, then we introduce a call to `llvm.loop.varbound` to continue iterating the correct number of times.

CHAPTER 6

Tests and Results

6.1 Unit tests

6.1.1 Backend

During the development, we used the LLVM test framework, `llvm-lit`. It allows us to write a program in LLVM IR, and specify a set of inputs and their associated expected outputs. Both input and output are an integer. Each unit test in LLVM in then run with different configurations (configurations modifies things such as the optimization level of the program, the activation of single-path specific optimizations or features). In total, there are 48 different configurations for each test (a detail of these configurations is provided in Appendix A), and these tests are run 5 times with 5 different inputs for each configuration. Additionally, different runs are compared between them, to ensure a constant execution time.

The interesting part is that we could set up these tests even before starting to build our solution. This aligns with the Test-Driven Development (TDD) approach[38], that consists of developing a new program in order to solve previously defined test cases, one test case at a time.

6.1.2 Frontend

Once our pragma has been implemented, we can also test it by compiling only from C source code to LLVM intermediate representation, and checking that the correct instructions have been attached to our loops. The testing here is simpler, as it is not run through different configurations (since it is only the frontend and that the actual transformation takes place in the backend). We use these tests essentially to verify that a pragma in our source code produces the correct call to `llvm.loop.varbound` in LLVM IR.

6.2 Benchmarks

6.2.1 TACLe

In many of the papers related to single-path code, the benchmark suite used was the TACLe benchmark suite, which consists of a collection of small C programs implementing simple algorithms (matrix multiplication, searching algorithms, car-window control system, etc.). This benchmark collection is used, for instance, in [39] or [13]. In order to actually run the programs, we use the Patmos Simulator, which can run programs compiled for Patmos and provide statistics about the program execution [8]. The key metric that we will use here is the number of cycles taken to execute our programs. To compile our programs, the only options that we use are :

- `-mpatmos-enable-cet` : To enable single-path code.
- `-mpatmos-cet-function` : That is needed to specify what the root function for single path will be.
- `-O2` : This is the default optimization level for tests.

However, this series of tests is not suited at all to our work, since most of the loops in these programs have a fixed number of iterations (i.e. Each loop will iterate N times, the upper and lower iterations number are the same). Since our work focuses on letting programmers define the upper bound as a variable, this is worthless. Appendix A details the situation of each program and how they fit and do not fit for our project.

While the nature of this test suite makes evaluating our solution complex, it can still be used to observe the impact on execution times. To do this, we replace one of the `loopbound` pragmas with a `varloopbound` pragma in the programs we want to test. In some cases, we may also modify the function containing the loop.

6.3 Artificial programs

We will now present several programs that we know are supposed to be improved by our work, and determine if this is effectively the case.

The first program, `bsort`, is a sorting program that is going to sort arrays of different sizes (one array per size is sorted), using bubble sort. The main sorting function takes a parameter specifying the size of the array for the new implementation. For the implementation, we specify the loop bound as `#pragma loopbound min 10 max MAXSIZE`, with `MAXSIZE` varying between different tests cases. Results are summarized in the Table 6.1 below.

```
int main() {

  bsort_Initialize(Array100, 100);
  bsort_Initialize(Array1k,  1000);

  bsort_BubbleSort(Array100, 100);
  bsort_BubbleSort(Array1k,  1000);

  ...
}
```

Listing 6.1: Parametrized bsort

The next program (`matrix1`) is performing a matrix multiplication. Originally, it only performs one multiplication between two 10 by 10 matrices. To test it, we parametrize the function effectively doing the multiplication, and run it two times, one with two 10x10 matrices, one with two 20x20 matrices.

```
int main( void )
{
  matrix1PinDown( &matrix1_A[ 0 ], &matrix1_B[ 0 ], &matrix1_C[ 0 ], 100); // 10*10
  matrix2PinDown( &matrix2_A[ 0 ], &matrix2_B[ 0 ], &matrix2_C[ 0 ], 400); // 20*20

  matrixMain(&matrix1_A[ 0 ], &matrix1_B[ 0 ], &matrix1_C[ 0 ], 10);
  matrixMain(&matrix2_A[ 0 ], &matrix2_B[ 0 ], &matrix2_C[ 0 ], 20);

  ...
}
```

Listing 6.2: Parametrized matrix1

Lastly, we can rewrite the test program cover.c, that is also part of the TACLe suite. This program assesses the coverage of switch statements. It does so by having three functions containing a for loop incrementing a variable called i from 0 to 10, 50 and 120 respectively. Then inside this for loop is a switch statement on the value of i, and for each value a global counter is incremented. At the end of the program, the counter is thus equal to $10 + 50 + 120 = 180$. The choice of having three functions instead of one function with a parameter for the number of iterations of the loop is done to optimize manually the program for single-path code. We can use our modification to parametrize this function and call it three time with the values of 10, 50 and 120, and use the new pragma to specify this new parameter as the loopbound. To benchmark `cover`, we compare to the parametrized function that could have been written before with a maximum number of iterations of 120.

The results for each program are given in table 6.1. For each program, we compute the performance increase between the old version and the new version using the formula :

$$\frac{oldVersion - newVersion}{oldVersion} \times 100$$

Program	Performance Increase (%)
bsort	45.0
matrix	48.9
cover3	12.8

Table 6.1: Benchmark Results

CHAPTER 7

Conclusion and Further Work

To conclude, we tried to solve the problem by using a constant propagation algorithm as well as our own analysis to identify and propagate constant variables used as loop bounds. This solution has been implemented in the LLVM-based Patmos Compiler. As a first step, the compiler was also modified to accommodate for the use of variables as loop bounds.

The fact that we are relying on LLVM's built-in optimization can be seen as a strong limitation, since some cases won't be handled. For instance, if an argument of a function is constant in one file, and used in another file, and that these two files are compiled separately and linked afterward. Despite this, our meticulous testing showed that our solution is effective for a wide variety of real-world cases. Furthermore, testing has shown that some performance improvements can be made where they were expected to be, with up to ∼49% speedup for ideal cases.

Another aspect of our work that could be improved is the way we are dealing with LLVM's intrinsic. Currently, when we are meeting a `memcpy` intrinsic, we just replace it with a loop. What could be done instead is defining a built-in function for the compiler and replacing the intrinsic call with this function using the new pragma.

Another aspect of this work that could be improved is the implementation of our solution. There are indeed a few instances of duplication. For instance, we added a new pragma called `varloopbound`, but there is already a pragma called `loopbound`, and we could use this already existing pragma to also specify loop bounds depending on variables, which would require a change in the parsing of the pragma. Due to how the parsing of variable is implemented in the current pragma, it should not be difficult to implement this change. Another instance of such duplication is the function used to store loop bounds in the LLVM IR : `varloopbound`. The already existing function `llvm.loop.varbound` could be used here, as it has the same argument's list. This change would affect the lowering of instructions, but as for the previous one, it should not be too difficult to implement.

APPENDIX A

Tests

In this appendix, we provide details about the testing of our implementation with the LLVM unit testing framework. The first part is the list of the different options used to test our programs, the second part will be an explanation of our tests.

To test each program in LLVM, we have a test_matrix specifying options. This matrix is composed of two lists, and each element of the two list must be combined with every element in the other group.

The first list is composed of the following options, where the first element of the tuple is an option to llvm, and the second element is an option passed to the Patmos simulator.

- `("", "")` : Traditional Code
- `("-mpatmos-serialize=" + compiled + ".pml -mpatmos-serialize-functions=" + sp_root, "")` : Single-Path Code without dual issue
- `("-mpatmos-singlepath=" + sp_root + " -mpatmos-disable-vliw=false", "-D ideal")` : Single-Path Code with dual issue
- `("-mpatmos-singlepath=" + sp_root + " -mpatmos-enable-cet=opposite", "-D lru2")` : Constant execution time using opposite predicate compensation
- `("-mpatmos-singlepath=" + sp_root + " -mpatmos-enable-cet=counter", "-D lru2")` : Constant execution time using decrementing counter compensation
- `("-mpatmos-singlepath=" + sp_root + " -mpatmos-enable-cet=counter -mpatmos-cet-compensation-function=__patmos_comp_fun_for_testing", "-D lru2")` Constant execution time using decrementing counter compensation with pointer to specific function

- `("-mpatmos-singlepath=" + sp_root + " -mpatmos-enable-cet=hybrid", "-D lru2")` : Constant execution time using heuristic choice between other algorithms

The second list is composed of the following options :

- `-O1` : Somewhere between no optimization and moderate optimization.
- `-O2` : Moderate optimization level.
- `-O2 -mpatmos-disable-pseudo-roots` : Disable pseudo-root functions.
- `-O2 -mpatmos-disable-countless-loops` : Disable countless loops optimizations.
- `-O2 --mpatmos-max-subfunction-size=64` : Low subfunctions size to make sure the splitter is working.

Here is a short description of some (the most significant) of our unit tests. It goes without saying that they all use our new pragma.

- Case 1: This simple test simply consists of a loop in the main function. This loop iterates v times, with x a global variable equal to 10
- Case 2: This simple test simply consists of a loop in the main function. This loop iterates x times, with x being an input argument of the program. Since we are not able to determine the maximum number of iterations of the loop, this program is expected to fail.
- Case 3: This test case has two functions, `main` and `add_to`. The `add_to` function takes three parameters, one of them (x) is incremented by a loop (each iteration adds one to the value of x), and one is the loop bound given in the pragma. In the `main` function, `add_to` is called with the variable x being an argument of the program, and constant integer values for the two other parameters of the function.
- Case 4: This test case has two functions, `main` and `add_to`. The `add_to` function takes two parameters, one of them (x) is incremented by a loop (each iteration adds one to the value of x), and the second is the number of iteration of the loop. The loop is given a loop bound value loaded from a global constant. In the `main` function, `add_to` is called with the variable x being an argument of the program, and constant integer values for the other parameter of the function.
- Case 5: This test case is similar to the case 3, except this time the call to `add_to` uses a variable for the loop bound. The only way for LLVM to generate such IR is that it cannot determine the loop bound argument to be constant when calling `add_to`, thus we expect this test to fail.

- Case 6: This test case only has a `main` function. In it, it checks if the input parameter x is odd. If it is odd, it defines the `result` variable to be equal to 8; otherwise, it sets it equal to 6. The program then enters a loop that accumulates the sum of a constant value (1) until it reaches the computed result. The loop bound variable used is the `result` variable defined earlier. Since its definition depends on the input of our program, we expect this test case to fail.

- Case 7: This test case is similar to tests cases 3 and 5, except this time the `main` function calls `add_to` two times, and each times it gives it constant value for the loop bound argument. This test case is an illustration of what is defined in the design as "Interprocedural Case 1".

- Case 8: This test case also consists of a `main` and a `add_to` function. Similarty to test case 4, `add_to` has two parameters. It is called two times (and both times with the loop bound argument given as an integer in the function call), but in two separate branches (thus it is not always called enabled). This is an illustration of what is defined as "Interprocedural Case 2", we are testing if our pass is able to determine the maximum loop bound.

- Case 9: This test case program defines two functions: `double` and `add_to`

  The "double" function takes an integer argument, called input performs two loops. The first loop accumulates the sum of the constant value 1 for input/2 iterations, and the second loop accumulates the sum of the constant value 2 for input/10 iterations. The final returned result is the sum of the results from both loops.

  The `main` function checks if its input x is odd. If true, it calls the `add_to` function with parameters (x, 9, 10) and stores the result in a variable. If x is not odd, it calls the "double" function with the argument 10 and stores the result in another variable. The final result is determined by the phi node based on the condition, and the overall result is returned.

- Case 10: This LLVM IR program defines three functions: `add_to`, `func1`, and `func2`, as well as the `main` function.

  The "add_to" function implements a loop that increments x by 1 for a total of *iterations* iterations, with the loop bound specified by the parameter *bound*. The main function calls `add_to` twice with different sets of parameters through the helper functions `func1` and `func2`. The final result, representing the accumulated sum of both function calls, is returned by the main function.

List of Figures

List of Listings

List of Algorithms

Bibliography

[1] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Springer US, Boston, MA, 1998.

[2] Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. ISSN: 2190-6807.

[3] Enrico Mezzetti and Tullio Vardanega. On the industrial fitness of wcet analysis. In *Proceedings of the 11th Int'l Workshop on Worst-Case Execution-Time Analysis*, 2011.

[4] P. Puschner and A. Burns. Writing temporally predictable code. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (WORDS 2002)*, pages 85–91, January 2002. ISSN: 1530-1443.

[5] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for Time Predictability. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 382–391, Berlin, Heidelberg, 2012. Springer.

[6] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, October 2015.

[7] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: a time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, April 2018.

[8] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. *Technical University of Denmark, Tech. Rep*, 2014.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[10] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, July 1970.

[11] Daniel Prokesch, Stefan Hepp, and Peter Puschner. A Generator for Time-Predictable Code. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 27–34, April 2015. ISSN: 2375-5261.

[12] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, September 2002.

[13] Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner. Constant-Loop Dominators for Single-Path Code Optimization. In *DROPS-IDN/v2/document/10.4230/OASIcs.WCET.2023.7*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2023.

[14] Joseph C. H. Park and M. Schlansker. On Predicated Execution. 1991.

[15] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004.

[16] Alfred V. Aho and Jeffrey D. Ullman. Optimization of Straight Line Programs. *SIAM Journal on Computing*, 1(1):1–19, March 1972.

[17] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated Static Single Assignment. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, page 245, USA, October 1999. IEEE Computer Society.

[18] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, May 2011.

[19] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction (extended version), March 2012. arXiv:1203.5303 [cs].

[20] Moritz Sinn and Florian Zuleger. LOOPUS - A Tool for Computing Loop Bounds for C Programs. In *EPiC Series in Computing*, volume 1, pages 185–186. EasyChair, June 2012. ISSN: 2398-7340.

[21] Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner. Compiler-Directed Constant Execution Time on Flat Memory Systems. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 64–75, May 2023. ISSN: 2770-162X.

[22] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, Lecture Notes in Computer Science, pages 104–113, Berlin, Heidelberg, 1996. Springer.

[23] Pornin, Thomas. SSL Git Repository. Available at `https://www.bearssl.org/gitweb/?p=BearSSL`.

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

[25] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[26] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, October 1973. Association for Computing Machinery.

[27] Keith D. Cooper and Linda Torczon. *Engineering: A Compiler*. Morgan Kaufmann, Amsterdam Heidelberg, 2nd edition edition, February 2011.

[28] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools: International Edition*. Pearson, Boston, Mass. Munich, 2 edition, February 2007.

[29] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[30] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 104–118, New York, NY, USA, January 1977. Association for Computing Machinery.

[31] Ben Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, SE-1(3):270–285, September 1975. Conference Name: IEEE Transactions on Software Engineering.

[32] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: a benchmark collection to support worst-case execution time research. January 2016.

[33] LLVM's Analysis and Transform Passes — LLVM 12 documentation. Available at `https://releases.llvm.org/12.0.1/docs/Passes.html#passes-sccp`.

[34] Castro-Godínez, Jorge and Daniel Moya. CES / clang-custom-pragma · GitLab. Available at `https://git.scc.kit.edu/CES/clang-custom-pragma`, August 2018.

[35] Guelton, Serge. Implementing a Custom Directive Handler in Clang. Available at `https://blog.quarkslab.com/implementing-a-custom-directive-handler-in-clang.html`.

[36] Simone Pellegrini. An experimental framework for Pragma handling in Clang. 2013.

[37] LLVM Language Reference Manual — LLVM 19.0.0git documentation. Available at `https://llvm.org/docs/LangRef.html#metadata`.

[38] Jeff Langr. *Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better*. Pragmatic Bookshelf, Dallas, Tex., 1st edition edition, November 2013.

[39] Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner. Towards Dual-Issue Single-Path Code. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 176–183, May 2020. ISSN: 2375-5261.