Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at
TU UB WIEN Universitätsbibliothek
The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Compilation for Predictable Real-Time Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ludwig Meier
Matrikelnummer 0526541

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Wien, 7. September 2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Compilation for Predictable Real-Time Systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Ludwig Meier
Registration Number 0526541

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Vienna, September 7, 2014 _______________________ _______________________
(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ludwig Meier
Klimtgasse 6/6/6, 1130 Wien

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 - einschließlich 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.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagung

Zuallererst danke ich meiner Partnerin Christiane, ohne deren Unterstützung und Motivation ich diese Arbeit wohl nie abgeschlossen hätte. Ebenso danke ich meiner Familie für Ihre Unterstützung. Meine Cousins Josef und Ferdinand, die zur selben Zeit mit mir studierten, waren mir mit ihrem Erfahrungsschatz eine große Hilfe während meines gesamten Studiums.

Außerdem danke ich Herrn Puschner für die Möglichkeit meine Diplomarbeit zu einer derart spannenden Thematik zu verfassen.

Abstract

Real-Time systems require guarantees on the execution time of the executed programs, commonly at least a Worst-Case Execution-Time (WCET) bound. The complexity of today's computing architectures makes it difficult to predict the execution timing of a program.

The single-path programming scheme improves timing predictability by eliminating the selection of different execution paths depending on the input data provided to the program. Such programs execute the exactly same sequence of instructions for every invocation.

This document describes the automated transformation to a single-path program from annotated source code. The transformation consists of a dataflow analysis to identify the input-data dependent branches in the program and a transformation phase that replaces these branches by utilizing conditional execution, i.e., predicated instructions.

Kurzfassung

Echtzeitsysteme benötigen Garantien zum Laufzeitverhalten der ausgeführten Programme, üblicherweise zumindest eine Schranke der maximalen Ausführungszeit (WCET). Die Komplexität heutiger Computer-Architekturen macht es allerdings schwierig das Laufzeitverhalten eines Programmes vorherzusagen.

Die Single-Path Programmier-Methodik vereinfacht die Vorhersage des Laufzeitverhaltens dadurch, dass vermieden wird, dass Eingangsdaten die Auswahl des Ausführungspfads beeinflussen. Derartige Programme führen bei jeder Ausführung exakt die gleiche Befehlssequenz durch.

Dieses Dokument beschreibt die automatische Übersetzung von annotiertem Quellcode in ein Single-Path Programm. Diese Übersetzung besteht aus einer Datenflussanalyse zur Bestimmung der eingangsdatenabhängigen Verzweigungen und einer Übersetzungsphase, die diese Verzweigungen durch bedingt ausgeführten Maschinencode ersetzt.

Contents

1 Introduction **1**

2 Theory **3**
2.1 Single-Path Code 3
2.2 Single-Path Conversion 4
2.3 Single-Path Conversion from a CFG Perspective 4

3 Input-Data Dependencies **23**
3.1 Dataflow 23
3.2 Determining Input-Data Dependencies 30
3.3 Annotations 42

4 Transformation **45**
4.1 Transformation Overview 46
4.2 Loops 47
4.3 Branches 66
4.4 Function Calls 87
4.5 Code Generation 91

5 Results **101**
5.1 Runtime 101
5.2 Cost Drivers 109

6 Conclusion and Outlook **113**

A Appendix **116**
A.1 Call Graphs 116
A.2 Execution-Time Distributions 117
A.3 Listings 125

Bibliography **137**

CHAPTER 1

Introduction

Real-Time systems require guarantees on the execution times of the programs executed. A common requirement for a program executed by a real-time system is the knowledge about its Worst-Case Execution-Time (WCET) bound. Only then it can be guaranteed that the real-time system meets its timing requirements. The complexity of today's computing-architectures makes it difficult to predict the execution timing of a program.

The single-path programming scheme improves timing predictability by eliminating the selection of different execution paths to depend on the input data provided to the program. It does so by replacing input-data dependent branches by input-independent branches and constant-time conditional expressions. The constant-time conditional expression allows to assign a condition, which controls the execution, to a program expression. The execution of the constant-time conditional expression is expected to take a single, constant execution time. As a consequence the execution time is predictable.

This document describes the automated generation of a single-path program from annotated source code. The single-path code generation is integrated into the LLVM compiler. Since the ARM[1] architecture targeted by this work does not immediately provide machine instructions forming constant-time conditional expressions, the transformation is implemented by using conventional conditional execution. The conditional execution used is not guaranteed to execute in constant time, therefore the resulting program is neither guaranteed to execute in constant time but the execution-time predictability is improved.

The transformed programs simplify the WCET-analysis since only a single execution path has to be considered in the analysis. Since this single execution path does not differ amongst program executions, timing anomalies [48] cannot arise from differing instruction-fetching patterns. And even without the availability of a constant-time conditional expression the absolute execution-time jitter is reduced.

Chapter 2 gives an introduction to single-path programs and explains how any WCET-bounded program may be automatically transformed into a single-path program.

[1]Acorn RISC[2] Machines, later Advanced RISC Machines

As a prerequisite for the transformation the input-data dependent branches need to be identified. Therefore a dataflow-analysis pass has been implemented, which is described in Chapter 3.

The implementation of the single-path transformation is shown in Chapter 4. The initial transformation steps, in the target-independent portion of the compilation process, prepare the loops and branches so that in a later, target-specific transformation pass, it is easier to generate single-path code.

In Chapter 5 the impact this transformation has on the execution time of the transformed programs is evaluated.

CHAPTER 2

Theory

Contents

2.1 Single-Path Code **3**
2.2 Single-Path Conversion **4**
2.3 Single-Path Conversion from a CFG Perspective **4**
2.3.1 Reducible Control Flow 5
2.3.2 Optimizations 14
2.3.3 Irreducible Control Flow 20

2.1 Single-Path Code

Single-Path code [35] has the key property to show the same execution trace for each execution. In conventional programs the actual execution trace depends on the input data provided to the program. In single-path programs a unique execution trace exists, that is executed independent from the provided input data. This is achieved by replacing branches that depend on input data by input-data independent branches and the use of conditional expressions.

For timing predictability the use of a constant-time conditional expression [35] is suggested. The constant-time conditional expression is, as its name implies, expected to execute in constant time, independent from the actual condition value, and therefore has predictable execution timing.

2.2 Single-Path Conversion

The single-path conversion is a transformation scheme to generate single-path code from any WCET[1]-bounded program. A detailed set of transformation rules for a high-level language has been presented in [36]. These transformation rules are reproduced in Table 2.1.

Construct S		Translated Construct $SP\llbracket\ S\ \rrbracket\sigma\delta$
S	`if` $\sigma = T$	S
	otherwise	$(\sigma)S$
$S_1; S_2$		$SP\llbracket\ S_1\ \rrbracket\sigma\delta;$ $SP\llbracket\ S_2\ \rrbracket\sigma\delta$
`if` *cond* `then` S_1 `else` S_2	`if` $ID(cond)$	$guard_\delta := \sigma;$ $SP\llbracket\ S_1\ \rrbracket\langle\sigma \wedge guard_\delta\rangle\langle\delta+1\rangle;$ $SP\llbracket\ S_2\ \rrbracket\langle\sigma \wedge \neg guard_\delta\rangle\langle\delta+1\rangle$
	otherwise	`if` *cond* `then` $SP\llbracket\ S_1\ \rrbracket\sigma\delta$ `else` $SP\llbracket\ S_2\ \rrbracket\sigma\delta$

`while` *cond*
`max` N `times`
`do` S
	`if` $ID(cond)$	$end_\delta :=$ `false`
		`for` $count_\delta := 1$ `to` N `do begin`
		$SP\llbracket$ `if` $\neg cond$ `then` end_δ `:= true` $\rrbracket\sigma\langle\delta+1\rangle;$
		$SP\llbracket$ `if` $\neg end_\delta$ `then` $S\ \rrbracket\sigma\langle\delta+1\rangle$
		`end`

	otherwise	`while` *cond* `do` $SP\llbracket\ S\ \rrbracket\sigma\delta$
`call proc` $p(pars)$	`if` $\sigma = T$	`call proc` $p(pars)$
	otherwise	`call proc` p`-sip`$(\sigma,\ pars)$
`def proc` $p(pars)$ S		`def proc` $p(pars)\ S;$ `def proc` p`-sip`$(pcnd,\ pars)$ $SP\llbracket\ S\ \rrbracket\langle pcnd\rangle\langle 0\rangle$

Table 2.1: Single-Path Transformation Rules, taken from [36] p. 388

The rules presented in Table 2.1 show a mapping from source constructs S to the single-path transformed construct $SP\llbracket\ S\ \rrbracket\sigma\delta$. Whereby σ denotes a boolean execution condition and δ is used to create a unique numbering for the different execution conditions. The function $ID(cond)$ maps to *true*, when the condition *cond* depends on input data, to $false$ otherwise.

2.3 Single-Path Conversion from a CFG Perspective

This section takes an alternative view of the single-path transformation. Instead of the procedural approach from Section 2.1 the observations described in this section are based on the control-flow graph representation of a program.

[1]Worst-Case Execution Time

Motivation As stated in [35] the single-path conversion is applicable to any program for which a WCET bound is computable, i.e., an upper bound for the number of loop iterations can be determined. This section should clarify why this is the case by showing a transformation that is applicable to all programs which have reducible control flow. The remaining programs, having irreducible control flow, may be transformed into reducible ones in a preprocessing step by using an algorithm like node splitting.

2.3.1 Reducible Control Flow

In this section it is shown how the single-path transformation [35] can be applied to programs with reducible control flows. In the conventional description of the single-path transformation the rules used to describe the transformation are based on an imperative language. These rules can be found in [36] on Table 1. The CFG[2] representation has the advantage of a much simpler structure than the source-code representation. Additionally, when restricting to reducible control flows, it has been shown that the entire CFG can be processed by two transformations T_1 and T_2. The transformations SPT_1 and SPT_2, which are based on T_1 and T_2, are described below. The original description of if-conversion is also limited to reducible control flows as described in the section about backward branches in [3] on page 183 ff.

Reducible graphs [1] have the property that the repeated generation of maximum intervals [6] (sometimes called Allen-Cocke intervals) and the replacement of these intervals by a single node leads to a graph with only one node remaining. An alternative reducibility test is described in [42].

Reducible graphs also have the property of collapsibility [18].

Collapsibility Collapsibility represents the fact that repeated application of the two transformations T_1 and T_2 results in a single remaining node on reducible graphs. The following paragraphs give a short summary of these transformations. For the detailed description please refer to [18] on page 249.

> T_1 = *removal of self-loops*
>
> T_2 = *collapsing of a node with a single direct ancestor into that ancestor*
>
> ---
> Collapsibility transformations as defined in [18] on page 249

The transformations T_1 and T_2 are shown in a graphical notation in Figure 2.1a and Figure 2.1b.

[2] Control-Flow Graph

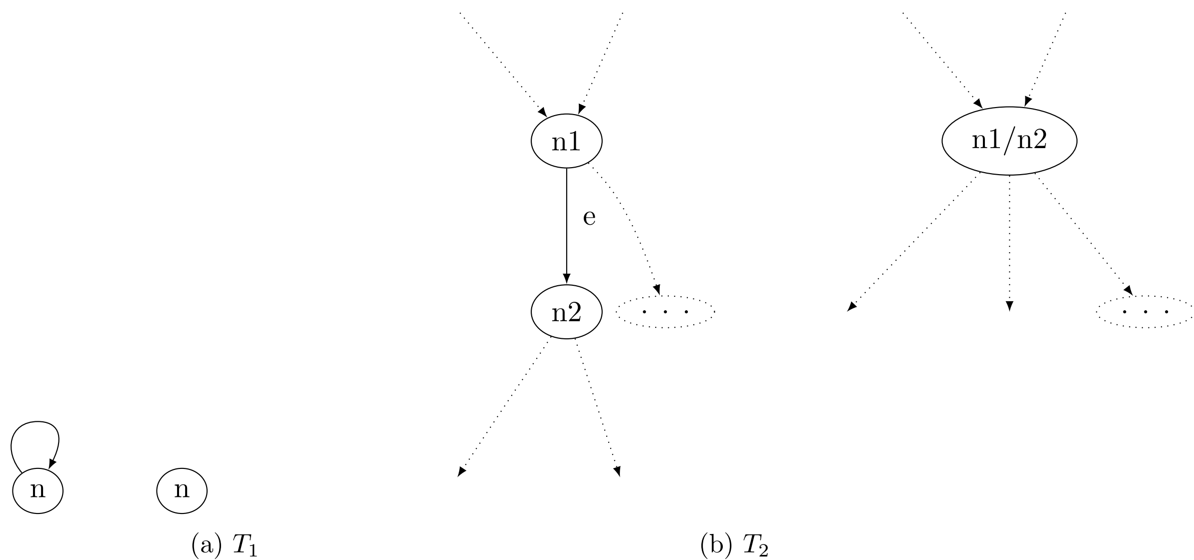


Figure 2.1: Collapsibility transformation rules

Modifying the Collapsibility Test to Determine Execution Conditions

Here a modified collapsibility test is used to show that it is possible to determine execution conditions for all nodes in reducible control-flow graphs.

The Results The algorithm presented here yields a modified CFG that is branch free with the exception of loop-back-edge branches. Essentially creating a chain of all the nodes in the CFG. Additionally an execution condition is calculated for each CFG node. These conditions compensate the fact that the execution of the program with the modified control flow will process nodes that it would not in the original program.

The Algorithm The modified collapsibility test $SPT()$ is shown as Algorithm 2.1. As the original variant it repeatedly executes 2 sub-procedures. These sub-procedures $SPT_1(n)$ and $SPT_2(n_1, n_2)$ are presented in Algorithm 2.3 and Algorithm 2.4.

As a preprocessing step for the following algorithm all loops must have been modified so that they have exactly one back edge. This may be achieved by inserting a new loop-latch node.

Then the modified collapsibility test is applied to the CFG. It computes an execution predicate for every node in the CFG. After the modified collapsibility test has been applied, for every CFG node n the variable σ_n holds the predicate under which n shall execute. When the reducibility test completely reduces a graph, every node except the initial node must have been used as node $n2$ on transformation T_2. Since the modified variant of T_2 does the predicate calculation, every node except the initial one has an execution predicate calculated. The initial node's predicate will remain *true* unless it is also a loop-header node and has therefore a loop-termination predicate assigned. Since the program execution always enters a function in its entry point represented by the

initial node in the CFG this node will always be executed. Therefore no execution condition is expected for the initial graph node.

The following algorithm analyzes the conditions that determine which control-flow path is taken during program execution. Assuming that two nodes u and w are adjacent, for any edge from node u to node w, C_{uw} denotes the condition that must hold for the edge from u to w being executed. When u has only one outgoing edge, $C_{uw} \equiv \top$. Note that C_{uw} only contains the conditions local to the basic-block u, i.e. the condition required for node w to be executed if node u is executed.

Note that parts of the algorithm that are tagged with *[code]* are not executed with the algorithm, but denote additions to the transformed program's code.

Algorithm 2.1: SPT(G)

Input : A reducible program with CFG $G = (N, E)$
Output: A CFG $G_{SP} = (N_{SP}, E_{SP})$ of a corresponding single-path program and a set of execution-conditions Σ_{SP} for the nodes in N_{SP}
Data: $N_{SP}, E_{SP}, \Sigma_{SP}$

```
1  SP_INIT();
2  while |N| > 1 do
3  |  if ∃e = ⟨n1 → n2⟩ ∈ E, n1 ≡ n2 then
4  |  |  SPT1(n);                              /* node with a self loop */
5  |  else if ∃n2 ∈ N, where n2 only has incoming edges from n1 then
6  |  |  SPT2(n1, n2);                /* node with an unique predecessor */
7  |  else
8  |  |  ERROR, input CFG is not reducible;
9  |  end
10 end
11 CreateLoopCounters();
12 return G_SP = (N_SP, E_SP), Σ_SP;
```

Analysis Results The results of the analysis presented in this section are:

N_{SP} The set of nodes for the single-path program. This is the same set as given in the input CFG.

E_{SP} The set of edges for the single-path program.

Σ_{SP} An execution condition σ_n for every node $n \in N_{SP}$.

The resulting CFG $G_{SP} = (N_{SP}, E_{SP})$ is a single-path program. The edges in E_{SP} form a chain with the nodes in N_{SP}. The only branches remaining in G_{SP} are those forming loops. When executing the program, the nodes in N_{SP} must only be executing when their corresponding execution condition in Σ_{SP} is met.

Initialization

Initially the condition σ_n for the start node is set to *true*, for each other node to $false$.

$$\forall n \in CFG : \sigma_n = \begin{cases} \top, & \text{if } n = \text{start node} \\ \bot, & \text{otherwise} \end{cases}$$

Additionally, each node n has a loop-membership label L attached, which is initialized to $\varnothing$, meaning that the node does not yet belong to any loop. This label is subsequently used to store the header node of the innermost loop that contains node n.

$$\forall n \in CFG : n.L = \varnothing$$

Further all graph edges are annotated by the labels S and T, whereby label S holds a reference to the edge's source node and label T that hods the edge's target node.

Algorithm 2.2: $\mathrm{SPT_{INIT}}$

```
1  Remove the edge n → n Introduce new loop-termination variable γ_h;
2  forall the nodes n ∈ CFG do
3  |   n.L ⟵ ∅;                                  /* loop membership */
4  |   if n ≡ START then
5  |   |   σ_n ⟵ ⊤;
6  |   else
7  |   |   σ_n ⟵ ⊥;
8  |   end
9  end
10 forall the edges e = ⟨u → w⟩, e ∈ CFG do
11 |   e.S ⟵ u;                                   /* source label */
12 |   e.T ⟵ w;                                   /* target label */
13 |   e.E ⟵ w;                                   /* exiting edge */
14 end
15 N_SP ⟵ ∅;                             /* nodes in the SP-Program */
16 E_SP ⟵ ∅;                             /* edges in the SP-Program */
```

$\mathbf{SPT_1}$

The transformation $SPT_1(n)$, as well as the original transformation $T_1(n)$, removes self loops of node n. The loop back edge $e_{back} = \langle n \rightarrow n \rangle$ at the application time of $SPT_1()$ may have different endpoints than the back edge had in the initial graph. The original start and end nodes are stored in edge labels. With the original start node $u \equiv e_{back}.S$ and the original target node $h \equiv e_{back}.T$

The loop is extended by an additional loop termination variable γ_h. The loop is considered active, as long as $\gamma_h \equiv \varnothing$. When the loop is left, γ_h identifies the edge through which the loop exited. The lines in Algorithm 2.3 that are prefixed with [*code*] are not considered to be executed with the algorithm. Instead the resulting program should be extended with these code snippets. These code snippets will manipulate γ_h at the program execution time.

The loop termination variable is initialized when the loop header h is entered from outside the loop. An implementation may append this assignments at the end of each node originating an edge incoming to h. Let E_{in} be the set of entering edges e_{in} targeting the loop header h excluding the back edge, with $i \equiv e_{in}.S$ and $h \equiv e_{in}.T$ being the edge endpoints in the original graph.

$$\gamma_h = \begin{cases} \varnothing, & \text{if } \sigma_i \land C_{ih} \\ \bot, & \text{otherwise} \end{cases}$$

Let E_{exit} be the set of exit edges e_{exit} leaving the node n except the back edge, with $u_e \equiv e_{exit}.S$ and $w_e \equiv e_{exit}.S$ being the edge endpoints in the original graph. The source node u_e of the exit edge is part of the loop, the node w_e is not contained within the loop. Set the loop termination variable to the following node outside the loop, whenever the loop is left through any of the exiting edges. For each edge $\langle u_e, w_e \rangle \in E_{exit}$, at the end of u_e set $\gamma_h = w_e$ when $C_{u_e w_e}$ is met.

For each node originating an exiting edge, store the loop membership in the loop tags attached to each node by setting them to the loop header. Existing loop memberships are not overwritten. This way the loop membership tag refers to the innermost loop containing the node.

$$\forall e \in E_{exit} : E.S.L \longleftarrow h$$

Set the execution condition of the loop header to regard the new loop termination variable. Since the loop header had the loop back edge as an incoming edge until now, it has not yet set another execution condition by SPT_2.

$$\sigma_h = \gamma_h \equiv \varnothing$$

Replace the back edge $u \to h$ by a input-independent loop enclosing u and w.

Algorithm 2.3: $\mathrm{SPT}_1(\mathrm{n})$

```
1  e_BACK ⟵ ⟨n → n⟩;                                        /* back edge */
2  h ⟵ e_BACK.T;                                          /* loop header */
3  l ⟵ e_BACK.S;                                           /* loop latch */
4  E = E \ e_BACK;                                 /* remove the edge e_BACK */
5  Introduce new loop-termination variable γ_h;
6  forall the edges e_in = ⟨i → n⟩, e_in ∈ IN(n) do
7  │ [code] γ_h ⟵ ∅;                   /* when the control flow follows e_in */
8  end
9  forall the edges e_EXIT = ⟨u_e → w_e⟩, e_EXIT ∈ OUT(n) do
   │ /* When this loop is terminated by e_EXIT                               */
10 │ [code] γ_h ⟵ w_e;               /* when the control flow follows e_EXIT */
11 │ if w_e.S.L ≡ ∅ then
12 │ │ w_e.S.L ⟵ h;                       /* innermost loop containing w_e.S */
13 │ end
14 end
   /* In case an edge originating from an inner loop also terminated
      this loop                                                            */
15 [code] γ_h ⟵ ⊥;                                       /* when ¬σ_l ∧ γ_h ≡ ∅ */
16 h.L ⟵ h;
17 σ_h ⟵ γ_h;      /* γ_h holds all conditions controlling loop execution */
18 E_SP ⟵ E_SP ∪ ⟨n.E → n⟩;           /* loop back edge in the SP-Program */
```

$\mathbf{SPT_2}$

Whenever the transformation T_2 is applied to the nodes n_1 and n_2, let E be the non-empty set of edges from n_1 to n_2.

The condition σ_{n_2} of node n_2 is set to:

$$\sigma_{n_2} = \sigma_{n_2} \vee \bigvee_{e(u,w)\in E} \begin{cases} \bot, & \text{if } w.L \equiv w, \text{ incoming edge is expected to set } \gamma_w \\ \gamma_{u.L} \equiv w, & \text{if } u.L \not\equiv \varnothing, \text{ loop exiting edge} \\ \sigma_u \wedge C_{uw}, & \text{if } u.L \equiv \varnothing \end{cases}$$

Note that the edges $e = \langle n_1 \to n_2 \rangle$ keep references to their original source and destination nodes in the labels $e.S$ and $e.T$ denoted as u and w here, even when these nodes got collapsed by an earlier application of SPT_2. In the example below these nodes are shown as labels next to edge endpoints.

Algorithm 2.4: SPT$_2$(n_1,n_2)

```
1  forall the multi-edges e ∈ E(n1, n2) do
2  │  u ⟵ e.S;
3  │  w ⟵ e.T;
4  │  if w.L ≡ w then
5  │  │  do nothing;
6  │  else if u.L ≢ ∅ then
7  │  │  σ_{n2} ⟵ σ_{n2} ∨ (γ_{u.L} ≡ w)
8  │  else
9  │  │  σ_{n2} ⟵ σ_{n2} ∨ (σ_u ∧ C_{uw})
10 end
11 forall the edges e = ⟨u, w⟩, e ∈ OUT(n2) do
12 │  E ⟵ E \ e ∪ ⟨n1 → w⟩;      /* copy labels from e to the new edge */
13 end
14 N ⟵ N \ n2;                                          /* remove n2 */
15 E_SP ⟵ E_SP ∪ ⟨n1.E → n2⟩;
16 n1.E ⟵ n2.E;
```

Evaluating the Execution Conditions

The execution conditions σ_n calculated for any node n are not globally valid. Instead they have to be evaluated right before node n is entered and must keep its value until the next execution of node n. Any further condition evaluation, that refers to σ_n, has to use this stored value because a re-evaluation could yield different results due to modifications to the variable space done in the meantime.

Update Control Flow When no further applications of SPT_1 and SPT_2 are possible, the applications of SPT_2 established a chain of nodes in E_{SP}. The only branches remaining are the loop-back edges introduced by SPT_1. These loops are transformed into loops executing for an input-data independent iteration number by the execution of $CreateLoopCounters()$.

In addition, the single-path transformation requires a so called constant-time conditional expression as presented in Section 3.1 of [35].

After application of the modified collapsibility test, the constant-time conditional expression can be applied to all nodes n except the new loop nodes created by *CreateLoopCounters()*, whereby σ_n is the condition that controls execution. The other side of the constant-time conditional expression remains empty.

Algorithm 2.5: CreateLoopCounters(N_{SP}, E_{SP})

```
1  forall the back-edges e_BACK = ⟨h → l⟩, e_BACK ∈ E_SP do
2  |  E_SP ⟵ E_SP \ e_BACK;                              /* remove e_BACK */
3  |  N_SP ⟵ N_SP ∪ h';                     /* create a new loop-header h' */
   |  /* Integrate h' into the control-flow                              */
4  |  e_o = ⟨l → n_o⟩ ⟵ OUT(l);   /* There is at most one outgoing edge */
5  |  E_SP ⟵ E_SP \ e_o;
6  |  E_SP ⟵ E_SP ∪ ⟨h' → n_o⟩;
7  |  forall the edges e_i = ⟨n_i → h⟩, l_o ∈ IN(h) do
8  |  |  E_SP ⟵ E_SP \ e_i;
9  |  |  E_SP ⟵ E_SP ∪ ⟨n_i → h'⟩;
10 |  end
11 |  E_SP ⟵ E_SP ∪ ⟨h' → h⟩;
12 |  E_SP ⟵ E_SP ∪ ⟨l → h'⟩;
   |  /* h' should take the branch h' → h as long as the input-data
   |     independent iteration bound is not hit.  Afterwards the edge
   |     h' → n_o is executed.                                          */
13 end
```

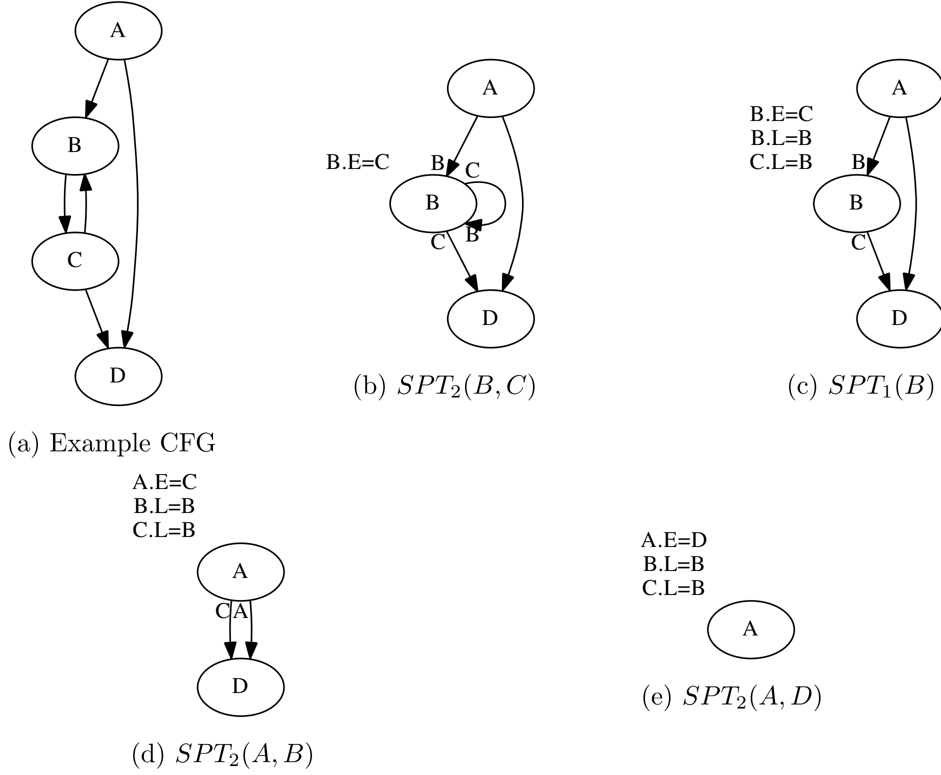
The Need for WCET Boundedness When transforming a program to an SP[3]-program, loops need to be modified so they show a unique iteration pattern regardless of the program's input data. This may be achieved by always iterating the loop for its iteration upper-bound. Such an upper bound must exist when the program is WCET-Bounded, although it may not be easy to determine.

Properties of the Resulting Control Flow The resulting control flow has no branches, except the ones introduced when transforming the loops. Since the loops are guaranteed to execute the same iteration count pattern for every call, the execution path for every call must be the same since no other branches exist in the program.

Example Transformation

The example in Figure 2.2 illustrates the application of the modified collapsibility test to a simple CFG.

[3]Single-Path



(a) Example CFG

(b) $SPT_2(B, C)$

(c) $SPT_1(B)$

(d) $SPT_2(A, B)$

(e) $SPT_2(A, D)$

Figure 2.2: Example application of SPT_1 and SPT_2

Transformation Step	σ_A	σ_B	σ_C	σ_D	addition to E_{SP}
	$\top$	$\bot$	$\bot$	$\bot$	
$SPT_2(B, C)$	$\top$	$\bot$	$\sigma_B \wedge C_{BC}$	$\bot$	B→C
$SPT_1(B)$	$\top$	$\gamma_B \equiv \varnothing$	$\sigma_B \wedge C_{BC}$	$\bot$	C→B
$SPT_2(A, B)$	$\top$	$\gamma_B \equiv \varnothing$	$\sigma_B \wedge C_{BC}$	$\bot$	A→B
$SPT_2(A, D)$	$\top$	$\gamma_B \equiv \varnothing$	$\sigma_B \wedge C_{BC}$	$(\gamma_B \equiv D)$ $\vee(\sigma_A \wedge C_{AD})$	C→D

Table 2.2: Calculation of the execution-conditions in the example shown in Figure 2.2

Update Control Flow The result of the analysis phase for the example given in Figure 2.2 are:
$N_{SP} = \{A, B, C, D\}$
$E_{SP} = \{B \to C, C \to B, A \to B, C \to D\}$
$\Sigma_{SP} = \{\sigma_A = \top, \sigma_B = \gamma_B \equiv \varnothing, \sigma_C = \sigma_B \wedge C_{BC}, \sigma_D = (\gamma_B \equiv D) \vee (\sigma_A \wedge C_{AD})\}$

In Figure 2.3 the graph is shown after having created the edges according to the analysis results.

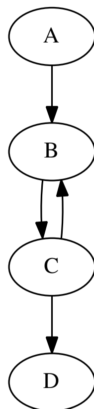


Figure 2.3: The example from Figure 2.2 before application of CreateLoopCounters()

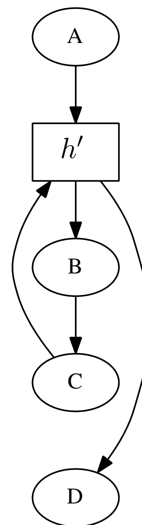


Figure 2.4: The result of applying CreateLoopCounters() to the CFG shown in Figure 2.3

When all the conditions are established, remove all branches except the loop back edges and introduce new branches as specified in E_{SP}, so that the nodes are lined up in the order they got merged by T_2.

2.3.2 Optimizations

In the following, modifications to the transformation are described which should reduce the execution runtime of the transformed program.

Application Order of SPT_1 and SPT_2

Strive to apply SPT_2 to nodes with the least possible application number of SPT_1. That means, for applications of SPT_1, that self loops are only removed for collapsed nodes that only consist of previously collapsed nodes belonging to the natural loop and no other nodes outside the loop. After the transformation is completed, nodes that are not part of the natural loop would require computing time with each loop iteration, whereby the execution predicates for these nodes will evaluate to $false$ for each except the last iteration.

With the SP-Transformation as described Algorithm 2.3 the application of this optimization is mandatory. This is because the resulting algorithm is simpler when the application order of SPT_1 and SPT_2 is constrained as described here. A modified version of the SP-Transformation that allows arbitrary application orders of SPT_1 and SPT_2 can be created by additionally tracking, for each node, the last node that has been col-

lapsed into it by an application of SPT_2. When SPT_1 is creating back edges as shown in Algorithm 2.3 on line 18 this additionally tracked node shall be used as the originating node of the back edge.

Keep Input-Independent Branches

Branches that do not depend on input data may be preserved by the transformation. Compared to the transformation shown in the paragraph *Update Control Flow*, these branches may skip some nodes in the CFG during execution.

Advantage As a result, when executing the transformed program, on occasion program parts are skipped by the preserved branches, resulting in a shorter execution time. Removing these branches instead and disable the effects of the now executed program parts by an application of the constant-time conditional expression to these program parts would yield the same result, but would make the program's execution more computationally expensive.

With the unoptimized transformation any node n of the CFG has to be executed, if the evaluation of σ_n immediately before the execution of n yields true during program execution. When the evaluation of σ_n yields $false$ and the evaluation result does not depend on input-data, the program flow still passes through n. In this case later transformation steps are responsible for isolating the modifications performed in n from the remainder program.

Correctness The optimized transformation presented here is guaranteed to not modify the program's behavior since all nodes that are effectively executed in the unoptimized variant are still executed in the same order. Therefore the preserved branches must not skip any nodes that should be executed, modify the execution order amongst executed nodes, modify the loop termination or introduce new loops.

Algorithm Modification Keeping some of the branches is done by a slightly different modification of the CFG than shown in the paragraph *Update Control Flow*. The modification of the control flow after the analysis phase starts by removing all branches except loop back edges and edges originating from an input-data independent branch. Otherwise the control-flow modifications are performed like shown in Algorithm 2.1. When the control-flow modification would insert an edge that has not been removed, the existing edge is kept.

Result The resulting CFG will at least contain all edges it would after applying the CFG modifications shown in Algorithm 2.1, but possibly containing some additional edges originating from nodes with input-data independent branches. Note, that the edges originating from these branches, which exist in both control flow variants, behave differently in the optimized variant. When their source node is executed, control flow is now only conditionally following these edges. Without the optimization the source node

would have had only one outgoing edge, therefore the control flow always had to follow this edge after executing the source node.

The calculation of the execution conditions has to be adapted to handle these new graph edges. Because nodes can be skipped, it is no longer guaranteed that for any node n the execution condition σ_p has been evaluated for every predecessor node p when the execution reaches node n. When such an edge is executed, it is obvious that any nodes skipped by this edge are not executed. Their execution conditions have therefore to be assumed to evaluate to $false$.

Skipped Nodes The nodes skipped can be derived by examining the applications of SPT_2 in more detail. The result is a set of skipped nodes $V_{skipped}(E)$ for every edge E originating from an input-independent branch.

Initially the set of skipped nodes $V_{skipped}(E)$ is empty for all edges E. Additionally for each input-independent branch b a set of all nodes targeted $V_t(b)$ by this branch is tracked, which initially contains all nodes targeted by outgoing edges from b.

When applying the modified collapsibility test, at some point there will be an application of $b' = SPT_2(b, n_{to})$ with b being an input-independent branch node and some other node n_{to} as the second. Remove n_{to} from the set of branch targets $V_t(b) = V_t(b) \setminus n_{to}$. The edge $b \rightarrow n_{to}$ is the outgoing edge from b that would also have been inserted in the unoptimized version of the transformation. When b has been collapsed by an application of SPT_1 or SPT_2 before, use the original branching block whenever referencing V_t .

For every application of $SPT_2(n, n_t)$ with $n_t \in V_t(b)$ and any node n, let N be the set of nodes collapsed into n. Add the nodes in N to the set of skipped nodes by the other outgoing edges from branching node b. When n is the branching node $n \equiv b$ or a node that “contains“ b, only add the nodes collapsed into n after b.

$$\forall n_o \in V_t(b) : V_{skipped}(< b, n_o >) = V_{skipped}(< b, n_o >) \cup N \tag{2.1}$$

The list of skipped nodes for the edge $b \rightarrow n_t$ is now complete, so remove n_t from the set of branch targets $V_t(b) = V_t(b) \setminus n_t$. When the set $V_t(b)$ is empty, the list of skipped nodes for all outgoing edges of branch b is complete.

Adapting Execution Conditions When all branch targets have been removed $V_t(b) \equiv \emptyset$, $V_{skipped}(n)$ contains the set of skipped nodes for every outgoing edge of the branching node b, with target node n.

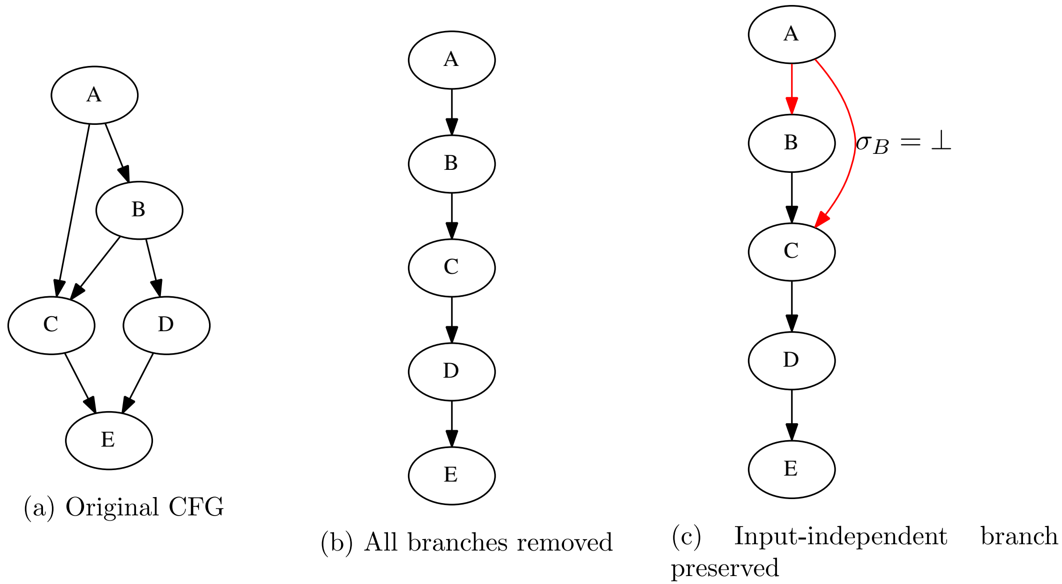
With the set of nodes skipped by an edge, one can think of executing the edge $E =< b, n_{to} >$ as setting the skipped nodes execution conditions to $false$, $\forall_{s \in V_{skipped}(n_{to})} : \sigma_s = \bot$, as shown in Figure 2.5c.

Alternatively, whenever any of the execution conditions refers to σ_s where node s is a node skipped by an edge $E =< b, n_{to} >$ branching from node b to the target node n_{to}, i.e. $s \in V_{skipped}(E)$, replace σ_s by $\sigma_s \wedge \neg(\sigma_b \wedge C_{bn_{to}})$. So, whenever σ_s would not have been initialized because it was skipped by the edge E, the expression $\neg(\sigma_b \wedge C_{bn_{to}})$ evaluates to $false$. In this case the conjunction with σ_s will always evaluate to $false$, independent from the value of σ_s.

Update Control Flow The control flow modification applied afterwards is very similar to the one presented in Algorithm 2.1, with some additional handling of the now preserved branches. When all the conditions are established, remove all edges except loop-back edges and edges originating from input-data independent branches. Introduce new edges so that the nodes are lined up in the order they got merged by SPT_2. For every application of $SPT_2(n1, n2)$ create an edge $n1.n_{exit} \rightarrow n2.n_{enter}$.

Example In the example shown in Figure 2.5a the branching condition in node A is considered being input-data independent, whereas the branching condition in node B is input-data dependent.

Figure 2.5b shows the result of an application of the control-flow modification that does not consider the input dependency of branches, as presented in Algorithm 2.1. In contrast Figure 2.5c finally shows the result of the optimized transformation shown in the previous paragraphs. The process of the execution-condition determination is documented in Table 2.3.



(a) Original CFG

(b) All branches removed

(c) Input-independent branch preserved

Figure 2.5: Control-Flow modification with preserved branches

	Without optimization	Branch preserve optimization
σ_A	$\top$	$\top$
σ_B	$\sigma_A \wedge C_{AB}$	$\sigma_A \wedge C_{AB}$
σ_C	$(\sigma_A \wedge C_{AC}) \vee (\sigma_B \wedge C_{BC})$	$(\sigma_A \wedge C_{AC}) \vee ((\sigma_B \wedge \neg(\sigma_A \wedge C_{AC})) \wedge C_{BC})$
σ_D	$\sigma_B \wedge C_{BD}$	$(\sigma_B \wedge \neg(\sigma_A \wedge C_{AC})) \wedge C_{BD}$
σ_E	$(\sigma_C \wedge C_{CE}) \vee (\sigma_D \wedge C_{DE})$	$(\sigma_C \wedge C_{CE}) \vee (\sigma_D \wedge C_{DE})$

Table 2.3: Execution-conditions calculation for the example shown in Figure 2.5

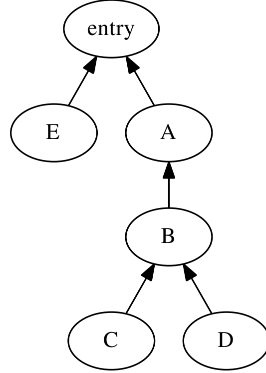


Figure 2.6: Control-Dependence Graph for the example shown in Figure 2.5a

Improved Application Order of SPT_1 and SPT_2 in case of Preserved Branches To reduce the programs execution time it is beneficial to skip as many nodes as possible with the branches that remain in the transformed program.

Again, assume an input-independent branch originating from node n_{branch}, with $V_t(n_{branch})$ being the set of nodes targeted by this branch. Let n'_{branch} be the node n_{branch} or a collapsed node containing n_{branch}. Similarly let n'_{to} be a node $n_{to} \in V_t(n_{branch})$ or a collapsed node containing n_{to}. Let $out(n)$ return the set of all outgoing edges of node n. $E_{out} = out(n_{branch})$ denotes the set of outgoing edges from n_{branch}.

The nodes that may potentially be skipped by an edge $e = n_{branch} \rightarrow n_{to}$, are at most all the nodes that are guaranteed not to be executed after the edge e was executed, without executing n_{branch} again, i.e., they may be executed in a future loop iteration. These nodes are a subset of the nodes that are control-dependent on n_{branch}. Skipping any other nodes that may later on turn out to require execution would require the introduction of additional back edges or additional executions of existing back edges.

The nodes that may be skipped are determined by the control dependencies in the program. Several different forms to express control dependencies have been described in the literature. The one that seams most convenient to determine the set of nodes that may be skipped is the set *cd* as defined in [31] on page 464 to determine this set of nodes. This set *cd* defines control dependencies on edges.

$V_{nx}(n, e)$ is the set of nodes that may not be executed when edge e is executed to leave node n. e is an outgoing edge of n_{branch}.

$$V_{nx}(e) = \bigcup_{e_{other} \in out(n) \setminus e} cd(e_{other}), e =< n, m > \tag{2.2}$$

In other words, when an outgoing edge e from some branching node is executed, all other outgoing edges will not be executed. As a result all nodes that would require any of the other outgoing edges being executed, will not be executed.

Let $V_{sk}(n)$ be the set of nodes that are skipped by all incoming edges to node n.

$$V_{sk}(n) = \bigcap_{e \in in(n)} V_{nx}(e) \tag{2.3}$$

Whenever selecting nodes for an application of SPT_2:

- Prefer $n''_{branch} = SPT_2(n'_{branch}, n_{sk}), n_{sk} \in V_{sk}(n'_{branch})$ whenever possible.
- Otherwise try to apply $n'_{branch} = SPT_2(n'_{branch}, n_t), n_t \in V_t$. Remove the edge from E_{out}, recalculate V_{sk}. Prefer n_t yielding maximal V_{sk}.

In Figure 2.6 the control-dependence graph for the example given in Figure 2.5a is shown. For the detailed definition of control dependence see [10] on page 323.

When comparing the control-dependence graph given in Figure 2.6 to the conversion result in Figure 2.5c one notices that only node B is skipped by the edge $A \rightarrow C$. Apparently this is only a very small subset of the set of nodes that are control-dependent on node A, which is $C_{dep}(A) = \{B, C, D\}$. Figure 2.7 shows the result of applying SPT_2 in an optimized order. The edge $A \rightarrow C$ is now skipping all nodes in $C(A \rightarrow B) \equiv \{B, D\}$.

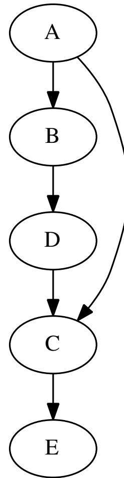


Figure 2.7: Improved application order of SPT_2

Optimized Control-Flow Modifications in case of Preserved Branches

The applications of the transformation SPT_2 impose a total order to all CFG nodes. The unoptimized variant of the transformation shown in section 16 is generating a program that executes all nodes in exactly this order. The control flow modification used in case of preserved branches as described in the previous paragraphs introduces some additional branches. The control-flow modification can take advantage of the additional control flow branches which result from keeping input-data independent branches. In the previous transformation variants the control flow was always transferred to the next node in order, when a node was left. Since a node may now be skipped by the additional

branches, occasionally it may be legal to transfer the control flow to a node further away in node order. This is the case when the following node is targeted by one of the additional branches in every case its execution is required.

Algorithm Modification When introducing the new edges in the application order of $SPT_2(n1, n2)$ and $n2$ is one of the nodes targeted by an input-data independent branch, for the first application with node $n1$, remember $n1$ and the original branching node b and create the edge $n1.n_{exit} \rightarrow n2.n_{enter}$ as above.

For further applications of $SPT_2(o, p)$, with p being another target node of the branch from $n1$, do not create the edge. Instead remember o. Note, there is already an edge from $b \rightarrow p$, this is one of the unconditional branches that has not been removed before.

Check if node p may not be executed if the branch $b \rightarrow n2.n_{enter}$ was executed, by using control dependencies as done to determine V_{sk} in Formula 2.3. Repeat this check in the order the nodes are connected until the first node n_x, for which this condition does not hold, is encountered. Create the edge $o \rightarrow n_x$. If the end of connected nodes is reached, defer further checks until the node becomes connected to its successor. Also consider nodes as connected when the connections get suppressed by this optimization. Obviously here is another optimization opportunity by choosing the application order of SPT_2 in a way that maximizes the number of successive nodes that match the above condition.

Example An example of this optimization is given in Figure 2.8a. This example is slightly extended compared to the previous ones to illustrate the different handling of input-data dependent and independent branches. The branch in node A is considered input-data independent, the branch in node C is considered to depend on input-data. Figure 2.8b shows the transformation result without the optimization described here. The optimized result is depicted in Figure 2.8c. The edge $D \rightarrow E$ is replaced by the optimized edge $D \rightarrow F$ because the node $E \in V_{sk}(D)$.

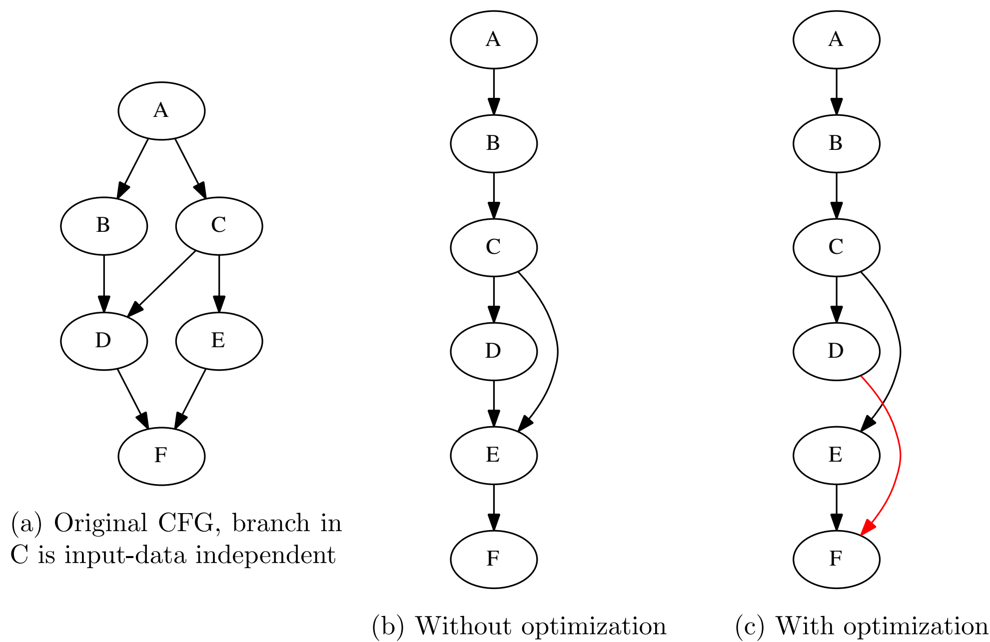
2.3.3 Irreducible Control Flow

The transformation shown in Section 2.3.1 is only applicable to reducible control flows as it is based on collapsibility and, as shown in section 5 of [18], only reducible graphs are collapsible.

Many real programs are reducibly anyway. In the 1970s Allen [2] found, for 75 FORTRAN[4] programs analyzed, that over 90% had reducible control-flows. Similarly Knuth (see [23] on page 110) analyzed a random sample of 50 FORTRAN programs out of 440 for reducibility, and found all of them to be reducible.

Fortunately, in a preprocessing step, every irreducible graph can be transformed into a reducible one [44]. One approach to convert irreducible graphs into reducible ones is based on node splitting.

[4]The IBM Mathematical FORmula TRANslating System



(a) Original CFG, branch in C is input-data independent

(b) Without optimization

(c) With optimization

Figure 2.8: Further reduction of executed nodes

Node Splitting A basic node splitting algorithm is described in [43] on page 817f. An optimized variant generating fewer nodes is presented in [44]. There they also have shown by the application of the optimized node splitting algorithm to several programs that node splitting was increasing the size of the resulting code between 2% and 12%. Whereby larger programs tended to show less relative growth.

Note however, that node splitting may come at the cost of an exponential growth in graph size for certain graphs, as shown in [5].

CHAPTER 3

Input-Data Dependencies

Contents

3.1 Dataflow **23**
3.1.1 Control Dependencies 24
3.1.2 Data Dependencies 24
3.1.3 Alias Analysis 25
3.1.4 Benefits of Context-Sensitive Alias-Analysis 26
3.2 Determining Input-Data Dependencies **30**
3.2.1 Data-Flow Analysis 30
3.2.2 Persisting Analysis Results 38
3.2.3 Input-Data Dependency as an IFDS Problem 39
3.3 Annotations **42**
3.3.1 Input-Data Dependencies 42
3.3.2 Loop Bounds 42
3.3.3 Functions to Ignore 44

3.1 Dataflow

The goal of the SP-Transformation is to remove input-data dependent branches from the program. Dataflow analysis allows the automated identification of these branches. Since the purpose of the SP-transformation is to generate a program whose execution traces are the same for each execution, input dependence in the sense of the SP-transformation refers to data that may differ amongst executions. Branches that are based on these data cause the execution trace to vary.

In the following a short overview over different program representations that can aid dataflow analysis is given.

3.1.1 Control Dependencies

During program execution, if a certain basic block is executed or not depends on the path selected by control-flow branches executed earlier. A basic block is said to be control dependent on those branches, which may affect the execution of this basic block.

> *Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff*
>
> *(1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and*
> *(2) X is not post-dominated by Y.*
>
> Definition of control dependence, from [10], page 323

Control dependencies can cause input dependencies as shown in the example in Listing 3.1. Assume x being input data. After execution of Listing 3.1 *isEven* holds a truth value depending on the value of x, without directly using x to calculate the value. Instead the two assignments to *isEven* are control dependent on the value of x.

```
1 bool isEven;
2 if(x % 2 == 0)
3   isEven = true;
4 else
5   isEven = false;
```

Listing 3.1: Control Dependence Example

Control-Dependence Graph

A control-dependence graph is a directed graph that contains vertices for each basic block and an additional root vertex. Edges go from controlling basic blocks to control-dependent basic blocks. Additional region nodes, as shown in [10], may summarize common control conditions.

3.1.2 Data Dependencies

The effect of program statements may, and usually does, depend on input data. Also statements may produce output data. Any two statements accessing the same data location are data dependent. An overview over different types of representations that may be used to represent data dependencies is given in the following paragraphs.

Dependence Graph

Dependence graphs [24] show dependence relations amongst program components. Besides loop dependencies, the dependencies covered by this type of graphs are caused by

read and write accesses to variables. For serially executed program statements, where statement i is followed by statement j the following dependence types have been identified in [24] on page 209.

Dependence Types:

- **Output Dependent** i and j write to the same variable.
- **Anti-Dependent** i reads a variable written by j.
- **Flow Dependent** A variable written by i is read by j, where no statement may modify the variable between execution of i and j.
- **Input Dependent** Both, i and j read the same variable.

A dependence graph is a directed graph with the program components as its vertices and directed arcs for dependencies between them.

Program Dependence Graph

The Program Dependence Graph [10] combines data- and control-dependencies into a single graph.

GSA-Form and the Program Dependence Web

The Program Dependence Web [30] is based on an SSA[1] representation of the program. It explicitly states control and data dependencies for each value calculated by adding gating functions. Annotating a program with only a subset of these gating functions, those corresponding to ϕ-functions in the SSA form, yields the GSA[2] form.

Thinned Gated Single-Assignment

The Thinned Gated Single-Assignment [13] form is like GSA also based on the SSA representation of a program, and extending that by gating functions. The TGSA[3]-form is defined for any programs with a reducible CFG.

3.1.3 Alias Analysis

When a program is in SSA-form, direct data dependencies are usually obtainable by querying the def-use chains. Def-Use chains allow to efficiently determine where the variables used by an instruction may have been defined. An exact definition of def-use chains can be found in [20]. The information of the input-dependence state of the definitions is then combined to derive the input dependence for the use, as is shown later on in Section 3.2.

When a program is not in pure SSA-form, as is the case for LLVM[4]s IR[5], not all data dependencies may be represented by def-use chains. In the case of LLVM arbitrary

[1] Static Single Assignment (form)
[2] Gated Single-Assignment
[3] Thinned Gated Single-Assignment [13]
[4] The LLVM-Project
[5] LLVM Intermediate-Representation

memory access is possible by the use of the load and store instruction, as is described in [27] on page 16. It is also possible to access memory locations that have not been explicitly declared earlier in the program. The dataflow analysis presented later on still requires to gather information about the data from these non-SSA memory locations. This is done by linking load instructions to the store instructions that may have written to the same memory location earlier by the use of alias analysis. Since alias analysis has been shown to be undecidable in the general case [25], the analysis will usually not be able to find the exact set of stores that may have created the value loaded at a certain program point. In these cases a safe over estimation of store instructions will be considered by the analysis implementation.

3.1.4 Benefits of Context-Sensitive Alias-Analysis

This section shows why the single-path transformation may benefit from context sensitivity in the data-flow analysis used to determine input-data dependencies. The context sensitivity may originate from the implementation of the data-flow analysis itself or from the usage of a context-sensitive alias-analysis.

With the results of a context-sensitive data-flow analysis the SP-Transformation may obtain differing input-data dependence information for differing call sites.

When context-sensitive results of the input-dependency analysis are available, the execution time of the transformed program may benefit from using that information by introducing branches that may skip parts of the code that would otherwise always be executed after SP-Transformation.

```
1  void foo() {
2    bar(exprInDep);
3    bar(exprNonDep);
4  }
5
6  void bar(int condition) {
7    if(condition) {  //input-dependent branch
8      expensive_calculation;
9    }
10 }
```

Listing 3.2: Example that may benefit from context-sensitive dataflow-analysis

```
1  void foo(bool pcnd) {
2    bar(pcnd, exprInDep);
3    bar(pcnd, exprNonDep);
4  }
5
6  void bar(bool pcnd, int condition) {
7    guard_1 = pcnd;
8    SP⟦expensive_calculation ⟧⟨pcnd ∧ condition⟩⟨2⟩;
9  }
```

Listing 3.3: Result of applying the SP-transformation to the code from Listing 3.2

The branch where context-sensitive data-flow analysis makes a difference when transforming the code from Listing 3.2 is the branch in line 7. It is input-data dependent when executing the call from line 2 and input-data independent when called from line 3, given that *exprInDep* is input-data dependent and *exprNonDep* is not.

In the following the effect of the single-path conversion to the execution trace of function *foo* as given in Listing 3.2 is examined. The transformation result is given in Listing3.3. When executing the function *foo*, with *exprInDep* and *exprNonDep* both evaluating to false, the execution trace would be 2, 7, 3, 7 before the single-path transformation is applied and 2, 7, 8, 3, 7, 8 after the application of the SP-Transformation. Note that before application of the single-path transformation the *expensive_calculation* is not executed and that after the single-path transformation *expensive_calculation* would be executed twice when only context-insensitive analysis-results are used. When the transformation implementation regards the context-sensitive data-flow analysis results it is able to create a program that has the run-time penalty of the SP-Transformation only in case the argument is actually input-data dependent. The transformed program may still contain the branch for the call with the input-independent argument *exprNonDep*.

Multiple Function Instances

As an optimization one could create different machine code that is used for different call paths reflecting the differing input dependencies calculated for different call paths. The result of applying this scheme to the example given in Listing 3.2 is shown in Listing 3.4.

This transformation could be added as a preprocessing step that does not interact with the SP-transformation. The following example shows the application at source level, but an implementation at the compilers IR level should also be possible.

To distinguish the function implementations generated for different sets of input-dependencies a string encoding the input-dependencies is appended to the function names.

To generate this string, first the list of all branches that are possibly input-data dependent on any call path, but are known to be input-data independent on at least one call path, is determined. When creating a function instance for a particular set of input-dependent branches B_{id}, for each branch in the branch list, I is appended to the function name if this branch is in B_{id}, otherwise N is appended.

```
1  void foo() {
2    barI(exprInDep);
3    barN(exprNonDep);
4  }
5
6  void barI(int condition) {
7    if(condition) {  //input-dependent branch
8      expensive_calculation;
9    }
10 }
11
12 void barN(int condition) {
13   if(condition) {  //non input-dependent branch
14     expensive_calculation;
15   }
16 }
```

Listing 3.4: Example that may benefit from context-sensitive dataflow-analysis

```
1  void foo(bool pcnd) {
2    barI(pcnd, exprInDep);
3    barN(pcnd, exprNonDep);
4  }
5
6  void barI(bool pcnd, int condition) {
7    guard_1 = pcnd;
8    SP⟦expensive_calculation ⟧⟨pcnd ∧ condition⟩⟨2⟩;
9  }
10
11 void barN(bool pcnd, int condition) {
12   if(condition) {  //non input-dependent branch
13     SP⟦expensive_calculation ⟧⟨pcnd⟩⟨2⟩;
14   }
15 }
```

Listing 3.5: Result of applying the SP-transformation to the code from Listing 3.4

The execution trace for a call of the function *foo* as given in Listing 3.4 is now compared to execution trace after the transformation. The resulting code from the SP-transformation is shown in Listing 3.5. When *exprInDep* and *exprNonDep* both evaluate to false the execution trace would be 2, 7, 13 before the single-path transformation is applied and 2, 7, 8, 13 after application of the SP-Transformation. Note that the *expensive_calculation* from line 14 is not executed in this case.

Optimization To reduce the number of method instances, exclude branches that do not result in runtime expensive code when SP-transformed from B_{id} and always SP-transform these branches. Also it is possible to create multiple instances only for those methods that appear most profitable. Possible measures for profitability may be the loop nesting depth of the method calls or the runtime profile of a previous execution.

Guarded Branches

As an alternative to the creation of multiple functions, it is possible to create branches that are used in the input-independent case only. When the branch is input-dependent predicated code would be used. Therefore it is necessary to distinguish if a branch is input-data dependent or not during execution. Given that this property will change for different executions of the same piece of code the input-dependency state is encoded within the data.

In the example given in Listing 3.6 the function is augmented by an additional parameter. This parameter carries the information if the sole branching instruction should be considered input-data dependent or not. In this example a boolean value *isID* is passed to the called function whose value is true if *condition* depends on input-data and false if not. The code resulting from an application of the SP-Transformation is shown in Listing 3.7.

At the call sites the value passed for this parameter will be chosen based on the dependency state for the corresponding branch. Of course the input-data dependency analysis requires to be implemented in a context-sensitive manner to yield differing result for the individual call sites.

The set of input-dependent branches that may have parameters added to function calls is the same than B_{id} in the previous paragraphs.

```
1  void foo() {
2    bar(true, exprInDep);
3    bar(false, exprNonDep);
4  }
5
6  void bar(bool isID, int condition) {
7    if(isID || condition) {  //considered non input-dependent
8      if(condition) {  //input-dependent
9        expensive_calculation;
10     }
11   }
12 }
```

Listing 3.6: Example that may benefit from context-sensitive dataflow-analysis

```
1  void foo(bool pcnd) {
2    bar(pcnd, true, exprInDep);
3    bar(pcnd, false, exprNonDep);
4  }
5
6  void bar(bool pcnd, bool isID, int condition) {
7    if(isID || condition) {  //considered non input-dependent
8      guard_1 = pcnd;
9      SP⟦expensive_calculation ⟧⟨pcnd ∧ condition⟩⟨2⟩;
10   }
11 }
```

Listing 3.7: Result of applying the SP-transformation to the code from Listing 3.6

The execution trace for a call of the function *foo* after the transformation as given in Listing 3.6 is now compared to execution trace before the transformation. When *exprInDep* and *exprNonDep* again both evaluate to false the execution trace would be 2, 7, 8, 3, 7 before the single-path transformation is applied and 2, 7, 8, 9, 3, 7 after the application of the SP-Transformation. Note that the *expensive_calculation* is only executed once. As a prerequisite it is required that the branch in line 7 has been preserved by the SP-Transformation. This is possible by forcing it to be considered not input-data dependent, regardless of its usage of the actually input dependent value *condition*.

3.2 Determining Input-Data Dependencies

To avoid the need of a manual attribution of each input-dependent branch in the program dataflow analysis is used to derive the input-dependent branches from annotations elsewhere in the program. This identification of input-dependent branches requires the support of a dataflow analysis implementation. The still necessary annotations of input-dependent data have to be placed in a manner that allows the dataflow analysis to derive input-data dependency for all input-dependent branches. Ideally the dataflow analysis is applied to the whole program, requiring annotations only at the program boundaries, i.e., wherever input-dependent data first is processed by the program. When applying the dataflow analysis to sub-portions of the program, e.g., on module level, the annotations have to be placed at least at all source locations where input data enters the unit of analysis.

This section describes the way used to determine which of the program's branches are input-data dependent and which are not.

The single-path transformation relies on the information which branches in the program are input dependent and which are not. Therefore an iterative data-flow analysis that determines for every instruction if it is dependent on input data has been implemented.

3.2.1 Data-Flow Analysis

To determine which branches in a program depend on input data, an iterative data-flow analysis at the level of the LLVM IR has been implemented. Within the LLVM instruction set the only way to access data from the system memory, be it on the heap or on the stack, is through usage of the load- and store instructions, as stated in [26], section 3.4. With these instructions it is necessary to rely on alias-analysis to determine possible data flows. This is because these memory accesses are not in SSA-form. The remaining data flow, which is in SSA form, may be deduced from the def-use chains and thus is a lot easier to handle.

The data-flow analysis is done in two steps. In a first step, detailed in paragraph *Summarizing the Effects of Function Execution*, the effect of each function to the set of input-data dependent memory locations is summarized. Building on these results in a second step, presented in paragraph *Iterative Analysis*, the iterative data-flow analysis is performed.

The architecture of LLVM enables easy global/inter-procedural analysis and optimization. For an overview description of LLVM's architecture refer to [26], section 2. Being able to do inter-procedural optimizations on the entire program was one of the original design goals of LLVM. The enabling part for the inter-procedural data-flow analysis is the LLVM linker which is provided with the LLVM set of tools. The LLVM linker allows linking of some/all modules available in LLVM IR, yielding one large IR module. The data-flow analysis is then performed on this linked module, thus avoiding any references to external functions that may not have been analyzed before linking.

Summarizing the Effects of Function Execution

The effect that calling a function has to the set of possible input-data dependent memory locations may be calculated once for each function. This calculation yields a mapping of function inputs to function outputs. In detail it is mapping inputs to those outputs that contain input-dependent data after function execution, when the input was input-data dependent.

An algorithm that yields similar results is presented in [38] with the method *BackwardTabulateSLRPs(WorkList)*. To obtain the same result the algorithm presented in the following can be executed once for every output of the analyzed function.

Analysis Order In the first analysis phase, the functions are evaluated in a bottom-up order as imposed by the call graph. That means that functions that do not contain function calls themselves are analyzed first. The current implementation does not support recursive function calls, which guarantees that the call graph is cycle free, so there always exist functions that do not call any other functions. From these functions on the call graph is processed towards the program entry, always analyzing functions that have all their callees already analyzed.

Inputs of a function considered by this analysis:

- The parameters passed to the function
- Load instructions

Outputs are:

- The function's return instructions
- Store instructions
- Arguments to calls of the functions *_ID* and *_NID*

The outputs that are in any case input-data dependent after a function call and those that are known to be not input-data dependent after a function call are also part of the analysis result. Those are stored in two sets separated from the other dependencies. So there are not special input symbols representing always-ID and never-ID that these outputs could depend on. And because both of these sets have the subsumption property as described in section 3.1 of [37], there are no other dependencies for outputs that are part of one of these two sets. Additionally, it is not possible for any output to be part of both sets at the same time.

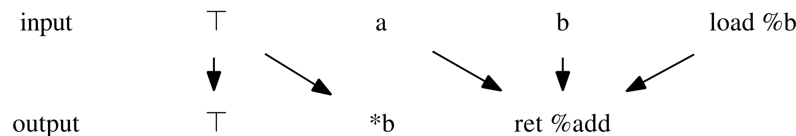
Graphical Notation The graphs given in Figure 3.1a and 3.2a show the inputs to a function on the top. The outputs are placed on the bottom. An edge from an input item to an output item means that this output has to be considered input-data dependent if the input may be input-data dependent. To keep the graphs simple, the outputs that are known to be always input-data dependent are shown as if there was a special input item *true* on which they depend. Any outputs that are known to not depend on any input data after the function execution are placed amongst the outputs but they do not have any incoming edges. Inputs that do not affect any outputs are not part of the graphs.

```
1 define i32 @callee(i32 %a,i32* %b){
2 entry:
3   %0 = load i32* %b
4   %call = call i32 @_ID (i32* %b)
5   %add = add nsw i32 %0, %a
6   ret i32 %add
7 }
```

Listing (3.8) Example Source to calculate a dependency matrix

```
1 int callee(int a, int* b) {
2   int r = *b;
3   _ID(b);
4   return a + r;
5 }
```

Listing (3.9) C-Source for Listing 3.8



(a) Dependency matrix calculated for Listing 3.8

Figure 3.1: Dependency matrix example

When a function has been analyzed there is no need to analyze this function again when another call to this function is encountered during creation of the dependency matrix of another function. Instead, the previously determined dependency matrix for the called function can be integrated by the application of simple rules. These rules are similar to those used in the iterative dataflow analysis as described in paragraph *Function Calls*, but instead of propagating input dependencies, the callee's dependence matrix is integrated into the caller's dependency matrix. Figure 3.2a shows an example combination of the dependency matrices.

Dependency integration rules:

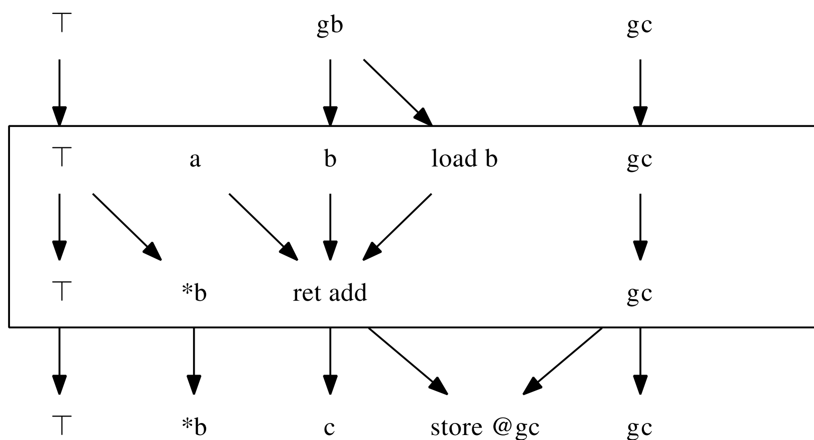
- Arguments passed to the callee are matched with the function parameters in the callee's dependency matrix.
- Load instructions in the callee are matched with any stores from the caller that may alias to them.
- When the callee's return value is used, the dependencies of all return instructions in the callee's matrix are added as dependencies of the return value.
- All other entries from the callee's dependency matrix are copied into the currently calculated dependency matrix.

```
1 define void @caller() {
2 entry:
3   %c=call @callee(int32 1,i32* %gb)
4    store i32 %c, i32* @gc
5    ret void
6 }
```

Listing (3.10) Calling function example

```
1 int gb, gc;
2 void caller() {
3   gc = callee(1, gb);
4 }
```

Listing (3.11) C-Source for Listing 3.10



(a) Dependency matrix calculated for Listing 3.10

Figure 3.2: Combine dependency matrix example

Calculation In a first iteration over all instructions all the outputs of a function are identified. These are the store and return instructions.

Then, for each of the outputs all instructions on which they may depend on are identified. These may be operands that are directly accessible through the use-def chain, or memory accesses yielding values that depend on previously executed store instructions when either the stored value is input-data dependent, or the execution of the store instruction itself is depending on some input data. The dependencies are located by

iterating through the instruction stream and the call graph in reverse order until no more dependencies can be determined.

When at least one of the dependencies is input-data dependent, it causes the output instruction to be considered input-data dependent when the program flow passes the analyzed function.

Arguments to function calls of _ID() are placed in the always input-dependent set, arguments to _NID() in the never input-dependent set.

Function Calls Whenever a function call is encountered during the backward walk through the CFG that means that this function may possibly be executed before the instruction whose dependencies are currently determined. The already calculated dependency information for this function is integrated into the dependency information that is currently calculated.

Control Dependencies Branches that impose a control dependency to basic blocks containing any of the output instruction, have all dependencies of these branch instructions added to the outputs. So, when a branch is input-data dependent, anything that is in the outputs of any basic block that is control dependent on this branch is marked input-data dependent.

Iterative Analysis

The second part of the dataflow analysis is implemented as an iterative worklist algorithm. The basic principle for this class of algorithms is described in [21]. The analysis implementation builds on the dependency matrices as described in paragraph *Summarizing the Effects of Function Execution* earlier in this section, to efficiently handle function calls.

Analysis Order To simplify the data-flow analysis, the programs supported by the implemented data-flow analysis are not allowed to contain recursive function calls. As a result the call graph is cycle free. The analysis starts at those functions that are not called from any other functions within the current module. When this analysis is applied to the whole program, after linking, this will usually be the program's entry point. From this top level function the analysis works down the call tree, always analyzing functions that have all their callers already analyzed.

It is possible to analyze the individual modules before linking, but since there may, and usually will, be function calls to functions within the analyzed module from some external module, input-data dependencies originating from these function calls will not be incorporated in the analysis result. As a result the analysis will underestimate the input-data dependent program locations.

Whenever a function call is encountered, the dependency mapping from the bottom-up analysis phase is used to determine how the function call would distribute the set of input-data dependencies that have been determined up to this point. This part of the

analysis phase requires some form of inter-procedural alias analysis. More details on this topic can be found in paragraph *Function Calls*.

Results The result of this analysis phase is a mapping of each IR-instruction in the program to an input-data dependence state. The analysis distinguishes 3 states of input-data dependence which are shown in Figure 3.3.

Initially all instructions have their input-dependence state set to *Unknown*. The analysis does then iteratively derive instructions that are known to not depend on input data or that may depend on input data. The analysis is completed and stops iterating when the input-data dependence states of all instructions have reached a fix-point, i.e., no more additional input dependencies can be determined.

Instructions for which it is not possible to derive an input-data dependence state, these are the instructions still *Unknown* when the analysis terminates, can be assumed to be input-data independent, given that all sources for input-data dependencies have been annotated and the whole program has been analyzed.

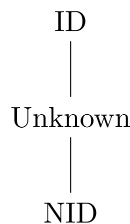


Figure 3.3: Hasse diagram of values applied during input-dependency analysis

The values from Figure 3.3 have the following interpretation:

ID May be input-data dependent.

Unknown May be either input-data dependent or not.

NID Known to not depend on any input data.

Note that the diagram given in Figure 3.3 differs by the additional value *Unknown* from the similar Hasse diagram given in [11], Section 2.3. Although the meaning of ID[6] and NID[7] are the same as in [11]. The additional state *Unknown* has no effect on the result of the analysis, because it is interpreted as NID when the analysis is complete. The benefit of tracking the *Unknown* state is purely in aiding the analysis implementation and debugging. So it is possible to distinguish values for which the analysis was able to deduce that on all code paths the value is known to be input-data independent because it only depends on constant values or values been annotated as input-data independent, and values for which no input dependency could be found on any program path.

[6]Input-Dependent

[7]Non input-dependent

Analyzing one Basic Block The dataflow analysis is modeled after the iterative algorithm shown in [21] and [43] page 763ff. For each basic block B a set of memory locations *out[B]*, that may hold input-dependent data after block B has been executed, is tracked. This set is initialized with the combined output of all predecessor blocks.

Let P be the set of predecessor blocks to B:

$$out[B] = \bigcup_{p \in P} out[p]$$

Iterating through all instructions, for each instruction in the block the appropriate dataflow function is applied to determine the effect that the execution of the instruction has on the state of input dependence. An instruction may cause additional input dependences, called *gen*. The *kill* set summarizes program locations which are known to be not input dependent after an instruction has executed.

The dataflow functions used in the iterative analysis must match the dataflow functions used to create the dependency matrices as shown in paragraph *Summarizing the Effects of Function Execution* earlier in this section. For the dataflow functions used to analyze the LLVM IR see Section 3.2.3. For an instruction I the *gen*- and *kill* sets can be derived by applying the appropriate dataflow function to this particular instruction. Let S be input-dependence state before the instruction, $S' = f(S, I)$ the corresponding gen-set is $gen[I] = S' \setminus S$ and the kill-set is $kill[I] = S \setminus S'$.

Gen Additions to the set of input-data dependent memory locations may be:

- Arguments to a call of the pseudo function *_ID()*.
- Target location of a store instruction, when the stored data may be input dependent.
- Input-Dependent stores within called functions or any stores within a function, if the execution of the function call itself is input dependent.

When a call of the function *_ID(id)* is encountered, *id* is considered a memory location that may carry input-dependent data from now on:

$$out[B] = out[B] \cup id$$

Kill Entries removed from the set of input-data dependent memory locations are:

- Arguments to a call of the pseudo function *_NID()* and any location currently in the set that is known to must-alias the argument to *_NID()*.
- Target location and any must-alias locations of a store instruction, when the stored data is not input dependent.

When a call to the function *_NID(nid)* is encountered, *nid* is considered as a memory location that does not carry input-dependent data from now on. From the set of input-dependent locations *nid* and anything that is known to alias to *nid* is removed:

$$out[B] = out[B] \setminus (nid \cup \{a | a \in out[B] \wedge mustalias(a, nid)\})$$

Function Calls Function calls are handled by integrating the information collected during the first part of the analysis presented above.

When a function call is control dependent on input-dependent data any side effect of the called function is added to current set of input dependencies. Even stores that are known to write input-independent data.

When the execution of the function call is not depending on input data, anything determined to be input-dependent in the intra-procedural analysis phase is added to the input-data dependent set. For every output of the function dependency matrix it is checked if any of the inputs is possibly input-data dependent. If so, this output is added to the current input-data dependent set. The information about stores that are known to store input-independent data is not used in the current implementation.

Propagation of Input Independencies up the Call Chain Currently no entries are removed from the set of input-data dependent locations when stores of input-independent data are executed within called functions. The reason for that is that none of the inter-procedural alias-analysis available to the analysis implementation will ever return always alias. Handling writes to global variables would be possible since global memory locations may be uniquely identified across function boundaries without requiring any alias analysis. Although this may be beneficial in some situations, it is not implemented either. So calls to *_NID()* inside functions are not propagated to the caller. The calls to the annotation function *_NID()* have to be repeated in the calling function's code to make it available to the data-flow analysis in the context of the calling functions.

Whenever the analysis passes a function-call instruction the dependency matrices, which are calculated as described in paragraph *Summarizing the Effects of Function Execution* earlier in this section, are integrated into the current output set.

Function Call Arguments For each function call, the arguments that may be input-data dependent are collected. When a function is called more than once from within the same caller function, all input-data dependent arguments are accumulated. When the function is analyzed later on, each formal parameter that has one or more corresponding arguments as part of the collected set is considered as input-data dependent.

Control Dependencies As soon as a branch is identified to be input-data dependent, all its control-dependent basic blocks are identified. Every store instruction in these blocks is then marked to be input-data dependent. Because they may not be executed depending on some input-data, it is not important if the value stored by these instructions is input-data dependent or not.

The same is true when the stores are part of functions that may be called from within any of these basic blocks, either direct or indirectly. So all the function calls in these basic blocks are evaluated as input dependent. This results in having all the callee's outputs added to the set *out[B]* of the calling basic block B. Luckily the function's dependency matrix already submerges the relevant information of all called sub-functions so no additional processing is required.

Finally ϕ-instructions may produce a value that depends on the branching instruction. These ϕ-instructions are identified by calculating parts of the TGSA-form as shown in [13] and [14]. In TGSA, ϕ-instructions are replaced by one or more gating functions. These gating functions do no longer depend on the incoming control-flow edge, but have an explicit predicate that selects the value they produce. ϕ-instructions with more than two incoming edges are replaced by a DAG[8] of gating functions integrated into the CFG at the appropriate control-flow join locations.

When the condition of the input-data dependent branch instruction is one of the predicates used to build the TGSA gating functions that would replace a ϕ-node, this ϕ-node is considered as input-data dependent. To determine if this is true for a particular ϕ-instruction it is not necessary to completely build the DAG of gating functions. Instead it is sufficient to calculate the set of selectors as described in [13], Section 4.2 and check if one of those blocks is terminated by an input-dependent branch.

Loop termination Loops are considered as input-data dependent, when their iteration count depends on input data. To determine if that holds for a particular loop the predicates calculated for the TGSA's η-nodes are examined. These predicates are constructed by *build_loop_predicate()* shown in [13]. The result of this procedure is a γ-DAG (stored in *Loop.Exits*) consisting of the conditions that contribute to the decision to terminate the loop. When any of the conditions in this DAG is input-data dependent, the loop is considered input-data dependent, which brings about that the appropriate SP-transformation will be applied to this loop in a later transformation step.

Marking instructions As the result of this analysis each instruction in the program is marked with one of 3 states reflecting its input-data dependency state. It may either be unknown if an instruction is input-data dependent, an instruction may be known to depend on input data or an instruction is known to not depend on input data. This mark is the sole interface linking the result of the data-flow analysis to the subsequent transformation passes.

3.2.2 Persisting Analysis Results

The *SP data dependence* analysis pass can store its analysis results along with the IR by using metadata. This is necessary, because some of the single-path transformation passes produce intermediate results that omit some control-flow information. E.g., some branches are replaced by increments to the guard variable. It is therefore not possible to derive the complete control-dependencies from the CFG after applying certain of the single-path transformation passes. One could in theory determine the control dependencies by analyzing the existing increments and decrements to the guard variable, but this has not been implemented. Instead, the *SP data dependence* analysis pass supports the argument *-run-analysis*, which instructs the pass to perform the data-dependency analysis and write the results to the metadata. All further usages of the *SP data dependence*

[8] Directed acyclic graph

analysis pass, without this argument, do not perform any analysis at all. Instead the result from a previous analysis run is read from the metadata.

Every instruction that depends on input data gets the metadata tag *!IDD* added. Every non input-data dependent instruction lacks this tag.

All the single-path transformation passes expect these metadata tags in place, so any of them has to be preceded by a run of the *SP data dependence* analysis pass along with the *-run-analysis* argument.

3.2.3 Input-Data Dependency as an IFDS Problem

The determination of those branching conditions that are input-data dependent may be achieved by solving an equivalent graph reachability problem. This is done by providing problem-specific dataflow functions that allow the construction of a so-called supergraph [37]. An analysis to a very similar problem to the input-data dependency analysis required for the SP-transformation has been done in [39] to identify program parts that depend on user input. Their intention was to use the information gathered by this analysis to further analyze the program for potential security vulnerabilities.

The approach of expressing inter-procedural dataflow analysis as graph reachability problems has been presented in [37]. There they have formalized the IFDS[9] framework and also provided the *Tabulate* algorithm, in section 4, which efficiently solves this class of problems. The key to being able to efficiently solve problems in the IFDS framework is that the dataflow functions are in $2^D \mapsto 2^D$, with D being a finite set. Because the dataflow functions in F are also required to distribute over the meet operator, i.e., the greatest lower bound, it is possible to deduce summary functions by combining several of the dataflow functions from F. Details why this is possible can be found in [37] on page 52f. The *Tabulate* algorithm is doing this whenever a function call is encountered. A summary function combining all data-flow functions for the entire procedure is calculated and stored in the set *SummaryEdge*. When further function calls to the same procedure are encountered, these calculations do not have to be repeated. Instead, the application of the summary function gives the same result. This is what makes this kind of analysis efficient and applicable to solve inter-procedural problems.

[9]Interprocedural,Finite,Distributive,Subset [37]

$$IP = (G^*, D, F, M, \sqcap)$$

An instance IP of an IFDS problem is formalized in Definition 2.4 on page 52 in [37]

G^* being the programs supergraph.
D being the union set of inputs and outputs as described above.
F a set of distributive functions.
M maps the edges in G^* to dataflow functions in F.
$\sqcap$ the meet operator is union.

Advantage over the Current Implementation

Running the dataflow analysis described in Section 3.2.1 should yield exactly the same results as the execution of the *Tabulate* algorithm shown in [37], whereby the later algorithm will show a better runtime behavior in cases where not all inputs to a function are input-data dependent. The *Tabulate* algorithm calculates the reachable outputs of a function for a given function input whenever this function input has been determined reachable from the program entry. The results of this calculation are then stored into the *SummaryEdge* Set. Any further function invocations of this function, with this particular function input being reachable, do not require further calculation, instead the results from the *SummaryEdge* Set are reused.

The implementation described above creates dependence matrices similar to those described in Section 3.2.1 that store the same information, but with the notable difference that these are calculated in advance to the main part of the analysis for all inputs of any function in the program. The *Tabulate* algorithm would only have to calculate the dependencies for inputs that are reachable for at least one invocation of a function. Function inputs that are not reachable, i.e., not input-data dependent in the context of the *SP data dependence* analysis, on any invocation of that function would never have their effects on the function outputs calculated.

Given that the dependence-matrix calculation as described in Section 3.2.1 is implemented in a bottom-up way, i.e., is determining for a given output of a function all the inputs that may cause it being considered input-data dependent, it is not possible to defer the calculation until a particular input is determined input-data dependent.

Dataflow Functions representing LLVM Instructions

In this section the dataflow functions for the LLVM-IR instructions are shown. These functions reflect how the execution of one of these instructions distribute input-data dependent values in the systems variable space and memory.

All functions are in $2^D \mapsto 2^D$ and following the notation in [37], S denotes a subset of 2^D. The functions $mayalias(p)$ and $mustalias(p)$ are required to return a subset of

2^D that contains all memory locations that may/must alias to p. Note that in any case p is part of the returned set. The function *controlling(inst)* returns the set of branching conditions controlling the execution of the instruction inst, which again is a subset of 2^D. Further C is used as a shorthand for the branching conditions controlling the execution of the current instruction.

Common Instructions For every LLVM instruction that is not explicitly listed below, the dataflow function is combining the instruction operands and control dependencies:

$$S = S \cup operands(inst) \cup C$$

call void @_ID(<ty>* %p) Calls to the marker function *_ID* are used to annotate input-data dependent memory locations in the source code:

$$S = S \cup mayalias(p) \cup C$$

call void @_NID(<ty>* %p) Calls to the marker function *_NID* are used to annotate memory locations that should be treated as input-data independent:

$$S = (S \cup C) \setminus mustalias(p)$$

store <ty> %v, <ty>* %p Store instructions may generate memory locations containing input-data dependent values by storing an input-data dependent value or by writing to an input-data dependent storage location :

$$S = \begin{cases} S \cup mayalias(p) \cup C & \text{, if } (v \in S) \vee (p \in S) \\ S \cup C & \text{, otherwise} \end{cases}$$

<ty> %v = load <ty>* %p The value returned by a load instruction is considered input dependent when a potentially aliasing store instruction may earlier have written an input-data dependent value, or when the memory location to load from is input-data dependent:

$$S = \begin{cases} S \cup v \cup C & \text{, if } (p \in S) \vee (mayalials(p) \cap S \neq \emptyset) \\ S \cup C & \text{, otherwise} \end{cases}$$

<ty> %v = phi <ty> [<val0>, <label0>], ... The value defined by a phi instruction is considered input-data dependent, when at least one of the alternatives is input-data dependent, or when the input value chosen is influenced by input data.

As a side note I'd like to state, that in this case there is no precision loss through the usage of SSA form. [29], Section 6 shows there could be, when only the incoming values of a phi instruction are considered and the control dependencies selecting the input value are not considered.

$$S = \begin{cases} S \cup v \cup C & \text{, if } (\bigcup_{n=0 \to N}(val_n) \cup controlling(\phi\text{-inst})) \cap S \neq \emptyset \\ S \cup C & \text{, otherwise} \end{cases}$$

3.3 Annotations

This section shows how source-level annotations are used to provide loop-iteration bounds to the SP-transformation. Further the annotations used to annotate the entries of input data into the program are described. These annotations are required by the *SP data dependence* analysis to identify the input-data dependent branches in the program.

3.3.1 Input-Data Dependencies

To indicate which memory locations contain input data and which do not, the calls to two marker functions are evaluated during the single-path transformation.

```
1 void _ID(void* p);
2 void _NID(void* p);
```

Listing 3.12: Function prototypes to annotate input data

A call to the function *_ID*, with a pointer *p* passed as the first argument, has the data in the memory location pointed by *p* considered input-data dependent. Any branches depending on data from this memory location will be transformed during the single-path transformation.

Memory locations that are known to not contain data that depends on input data can be attributed with calls to *_NID*. In the current implementation any *_NID* annotation is not propagated to the callers of a function. Therefore it may be necessary to declare a single memory location being input-data independent in several places.

3.3.2 Loop Bounds

A comparison of languages that are used to annotate loop bounds and other properties used in WCET determination is done in [22].

The annotation approach implemented is using annotation function calls. These calls to the annotation functions are manually placed to prepare the C source code for SP-transformation. Because these function calls are placed within the source code they are relatively easy to add and maintain. A benefit of this type of annotation is that the implementation does not require any modifications to the C-frontend. The annotation functions are compiled as any other function call would be. The evaluation of the annotation function calls is done entirely on IR-level, where most of the single-path transformation is done anyway. When the annotation calls are no longer required, these function calls are removed from the programs IR.

When the single-path transformation encounters a loop with a possibly input-data dependent iteration count but no loop-bound annotation, the file name and line number of the loop's source-code is reported in an error message, making it obvious where further loop-bound annotations are required.

Loop bounds are attributed as calls to the function *_LOOP_BOUND*. This call is placed right in front of the loop it belongs to.

The *_LOOP_BOUND* function declaration is expected to match:

```
1 void _LOOP_BOUND(unsinged int bound);
```

Listing 3.13: Function prototype to annotate loop bounds

The *_LOOP_BOUND* call is matched to a loop by walking up the chain of immediate-dominator blocks of the loop header. The instructions in each of these dominator blocks are scanned for a *_LOOP_BOUND* call. Searching for the loop bound stops when either a loop-bound is found or when the immediate-dominator blocks belongs to another loop than the one containing the loop-header from which the search started.

During the execution of the SP-transformed program, the argument to *_LOOP_BOUND* is evaluated once before the loop is entered. The constant bound returned by the function is then used to initialize a loop counter. This approach has been chosen because it is relatively easy to implement and also easy to debug. However it requires that the loop bound is calculated before the loop is entered.

Alternative approach

This section shows an alternate approach to loop-iteration bound annotations than the one implemented and shown in Section 3.3.2. Using this kind of annotation it would be possible to specify a loop bound that is determined iteratively. The single-path transformation in its current implemented does currently not support this type of annotation, but they may be a useful extension.

```
1 void _LOOP_BOUND_IT(bool belowBound);
```

Listing 3.14: Function prototype to annotate loop bounds that are determined iteratively

The argument *belowBound* reflects if the number of loop iterations performed to the current point of time is below the loop-iteration bound as it may be determined from input-data independent information. The argument to *_LOOP_BOUND_IT* would be evaluated once before every loop iteration. The evaluation result would determine if a further loop iteration has to be performed or if the loop is terminated since the loop bound is hit. To keep the single-path property, in SP-code the loop has to be iterated even when it has is already logically terminated. With these iterations the evaluation of the *_LOOP_BOUND_IT* argument has to take place. One has to take care that the evaluation of the loop-bound argument does not have any side effects modifying the program's behavior.

Listings 3.15 and 3.16 illustrate the usage of *_LOOP_BOUND* and *_LOOP_BOUND_IT*. This example is a slightly modified version of the example code used in [34]. It performs a binary search on an array of sorted integers. It assumes that a constant *SIZE* giving an upper bound for the parameter size is available.

```
int search(int k,int a[],int size)
{
  int l = 0, r = size-1, idx, inc;

  _LOOP_BOUND(log2(SIZE) + 1);
  while (r >= l) {
    idx = (r + l) >> 1;
    if (a[idx] == k)
      return idx;
    r = (k < a[idx] ? idx-1 : r);
    l = (k > a[idx] ? idx+1 : l);
  }

  return -1; //not found
}
```

Listing 3.15: Example loop with a bound annotation

```
int search(int k,int a[],int size)
{
  int l = 0, r = size-1, idx, inc;
  int bound = SIZE<<1;

  while (r >= l) {
    _LOOP_BOUND_IT((bound>>=1)>0);
    idx = (r + l) >> 1;
    if (a[idx] == k)
      return idx;
    r = (k < a[idx] ? idx-1 : r);
    l = (k > a[idx] ? idx+1 : l);
  }

  return -1; //not found
}
```

Listing 3.16: Bound annotation that is evaluated iteratively

3.3.3 Functions to Ignore

Specific functions can be attributed so that they are ignored by the single-path transformation. This is done by adding *__sp_ignore* to the function attributes. Clang, one of LLVMs C frontends, has been modified to understand this attribute. It is passing this attribute throughout the compilation stack so that it ends up as an additional function attribute in the LLVM-IR.

Functions with this attribute are ignored during the data-flow analysis used to determine the input-data dependencies. Nevertheless these functions may contain instructions that store input-data dependent values. To ensure that the remaining analysis steps correctly assume that these memory locations hold input dependent data, some additional annotations may be required.

CHAPTER 4

Transformation

Contents

4.1 Transformation Overview **46**
4.1.1 Execution Guard 46
4.2 Loops **47**
4.2.1 Notations 48
4.2.2 Input Dependency of Loops 49
4.2.3 Iteration Bounds 50
4.2.4 Transformation 51
4.3 Branches **66**
4.3.1 Notations 67
4.3.2 Branches to Transform 67
4.3.3 Preparation 68
4.3.4 Predicate Determination 70
4.3.5 Transformation 80
4.3.6 Reordering the Control Flow 83
4.4 Function Calls **87**
4.4.1 Passing the Guard Value at Function Calls 88
4.4.2 Implementation Details 89
4.5 Code Generation **91**
4.5.1 Code Generation Overview 91
4.5.2 Turning Branches into Predicates 94
4.5.3 Implementation 95
4.5.4 Transformation before Register Allocation 96
4.5.5 Pseudo Instructions 97
4.5.6 Transformation after Register Allocation 98

4.1 Transformation Overview

This chapter describes the implementation of the SP-transformation in detail. At first the handling of guard variables is shown in Section 4.1.1. The individual transformation steps described later on rely on this guard variable. The actual SP-transformation is split in 3 parts, described in the Sections 4.2, 4.3 and 4.4. Finally the modifications done to the code generation phase of the compilation process are shown in Section 4.5.

phi vs. ϕ The transformation and its accompanying analyses, which are described here, work, among others, on the SSA representation of a program. The transformation implementation is working with LLVMs IR which is also based on SSA form.

Throughout this document when referring to the general concept of SSA, ϕ is used. When referring to the `phi` instruction that is part of LLVM's IR, the word *phi* is used instead.

4.1.1 Execution Guard

In the process of the SP-transformation a large number of conditions are determined, which determine the execution of individual basic blocks. As it is later shown in Section 4.3.4 not all of these conditions have to be stored individually. Instead the condition portion that does not change over a SESE[1]-region may be extracted into its own variable. Further it is not required to store the conditions for each section separately. Since SESE-regions may not interleave, one storage location is sufficient as long as it is updated at the SESE-region's boundaries. This single storage location is called the guard or guard variable here.

Comparison of Storage Locations

The guarding variable will be frequently accessed by the SP-transformed program since it has to be updated on every input-dependent branch in the program. Further it is used to determine if input-dependent instructions should execute. Therefore the storage location of the guard variable will affect the runtime of the transformed program. In the following paragraphs the advantages and disadvantages of 3 different potential storage locations are evaluated. The current implementation provides support for switching between storage in a global variable and on the application's stack memory.

Memory location types that may hold the guard value:

- Global Variable
- Stack
- Register

Global Variable Storing the guarding value in a global variable results in the execution of many unnecessary load and store instructions because the common LLVM

[1] Single-Entry, Single-Exit

transformation passes may not completely optimize them away. Additional, before executing these load and store instructions, the address of the memory location containing the guard value must be loaded. A pointer to the global variable does not have to be passed with function calls, since global variables are also accessible, by name, from within the called function. Function signatures do not require any modification when passing the guard value in a global variable. During execution, functions may update the guard value at control flow branches to reflect the truth value of the branching condition. Any functions must restore the guard value when they return, since the value may be used by the calling function later on.

Stack Access to a stack variable may be faster, since stack offsets are smaller and may not require to be explicitly loaded into a register. Stack accesses are subject to optimization by LLVM's optimization pass $PromoteMemorytoRegister$ or calls to $PromoteMemToReg$. These optimizations may completely eliminate certain memory accesses, e.g., by keeping the guard value in a register when appropriate. When the guard value is located on the function's stack, it needs to be passed to any callee's requiring that value. It may be passed either as a parameter to called functions or otherwise in a global variable, if the function signature should not be modified. The called function is not required to restore guard value before it returns, because the callee is operating on a copy of the guard value.

Register One of the general-purpose registers could be reserved to hold the guard value. In this case the entire remaining code must never modify the guard register. Within the LLVM compilation architecture this could be achieved by modifying the hardware description used to generate the machine-code generator. The generated code would then be no longer ABI[2] compatible, if the code is generated by another compiler, which is not aware of this reserved register. Accessing the guard register would always be fast since there is never a requirement to access any memory. As with global variables any called functions will have to restore the guard value before they return, or the caller has to create a duplicate. Reserving one of the general-purpose registers would result in increased register spilling in non-guarding code due to the reduced number of available registers.

4.2 Loops

Single-Path code guarantees that every program execution produces exactly the same execution trace. This section covers the analysis and transformation necessary to guarantee the same execution traces with the existence of loops within the transformed program.

For loops that depend on input data, some kind of modification is required to guarantee input-data independent execution traces. This requirement is known since the first publications on single-path execution [35]. Also the basic transformation required

[2] Application Binary Interface

to guarantee input-data independent execution traces with loops has been described in [35], Section 3.3. The approach taken there was to convert the loop into one with a constant trip count and adding an additional condition to the loop body that represents if the original loop would already have terminated. Later it turned out that it is sufficient to use an input-independent iteration counter instead of a constant one [36], p. 385.

For a loop, to yield the same execution trace for several invocations, it is required to perform the exact same number of loop iterations. Also each iteration has to execute the same back edge as the matching iteration in an earlier or subsequent program execution. Finally the executed exit edge has to match across executions.

4.2.1 Notations

The following paragraphs give a short description of the notations used when describing loops. They are mostly from [2], [17] and [16], page 374.

Loop is a closed path in the CFG.

Loop Header is the unique entry node to a natural loop.

Pre-Header is a CFG node originating the unique edge from outside the loop to the loop header.

Loop Latch is a node in the loop which has the loop header as its immediate successor.

Entry Edge is an edge from a node not part of the loop to a node that is part of the loop. In reducible loops the only valid target node for an entry edge is the loop header.

Entry Node is a node in the loop with an immediate predecessor that is not part of the loop. For natural loops the unique entry node is called the loop header.

Exit Edge is an edge from a node within the loop to a node outside the loop. The originating node of an exit edge is called exiting node, the target node is called exit node.

Exiting Node is a node that is part of a loop and originates at least one edge targeting another node, an exit node that is not part of the loop.

Exit Node is a node not part of a certain loop that is targeted by an edge originating from another node, an exiting node, which is part of the loop.

Back Edge is a control-flow edge from a block contained in the loop to the loop header. With reducible flow graphs the set of loop back edges can be unambiguously identified [16]. Removing the set of back-edges from the control-flow graph leaves a DAG.

The basic blocks which are part of a natural loop are determined by the back edge. The loop consists of all nodes dominated by the loop header for which a path exists that contains the back edge but does not contain the loop header.

Single-Entry Loop is a cycle in the CFG with a single-entry node that dominates all other nodes in the cycle. In a reducible flow graph, any cycle is a single-entry loop. A single-entry loop consists of one or more natural loops.

Natural Loop is a loop which is defined by a single back edge. Several natural loops may share the same loop header node.

Natural loops are loops with a single entry block, the loop header, and have one back edge from a block within the loop to the loop header. The entry block is dominating all blocks within the loop. A detailed definition for natural loops can be found in [43] on page 738 and also in [16] on page 374.

Irreducible Loop is a loop for which edges exist that originate from outside the loop and target blocks in the loop.
Such non-natural loop cycles can occur only in irreducible flow graphs [2]. The control flow enters irreducible loops through more than one entry block. The set of back edges in the CFG is no longer unambiguous.

4.2.2 Input Dependency of Loops

This section describes how input-dependent loops are identified. Further details on input-data dependency can be found in Section 3.1.

The key property of a single-path program is the fact that its execution trace is always the same [35], regardless of the input data provided to the program. So the goal of this analysis is to identify those loops, whose execution trace depend on any input data. Those loops are called input-data dependent or input dependent for short.

For the purpose of single-path code generation a loop is considered as input dependent, when the loop termination depends on input data. This is the case, when at least one branch leaving the loop is input dependent. Any other branches are handled separately as described in Section 4.3.

As with any branch, data and control dependencies have to be considered when determining the state of input-data dependence for one of the loop exiting branches.

As noted in [10] on page 323, the decisions leading to the termination of a loop form a strongly connected region in the control-dependence graph. Since control dependencies propagate input-data dependencies, as described in Section 3.2.1, when the execution of one of the loop exits is input-data dependent, all other exits of this loop will also be considered input-data dependent by the analysis described in this section.

Iteration Branches

A possible approach to find input-dependent loops is to identify the iteration branches for a loop with the algorithm described in [15], Section 2.1. Iteration branches are the branches, which affect the number of loop iterations.

A loop is considered input dependent, when at least one of the iteration branches depends on input data. The iteration branches identified by this algorithm do not only contain loop exiting branches, but also branches that control the execution of these branches. In [15] this set of branches is used to calculate bounds for the loop iteration count. The same set of branches can be used to show the input dependence of the iteration bound by showing that any one of the branch conditions is input-data dependent. The set of iteration branches consists of the loop exiting branches and their control depen-

dencies within the loop. As noted in [10], page 323 these are the control dependencies forming an SCR[3] in the CFG.

TGSA to detect input-dependent loops

Using the information provided by TGSA [13] simplifies the identification of input-dependent loops by providing additional information compared to the SSA representation. TGSA annotates SSA by replacing the ϕ-instructions with any of the pseudo assignments γ, μ and η that carry additional information. The pseudo assignment's type is determined by the control flow as listed in the table below:
TGSAs pseudo-assignment placement:

- $\boldsymbol{\gamma}$: at control-flow joins of forward edges.
  They do, as ordinary ϕ nodes, hold a list of values from which one is selected. In addition a predicate is stated that determines which of the values is selected. With ϕ nodes one would have to analyze the control dependencies to obtain the information contained in this predicate.
- $\boldsymbol{\mu}$: at the loop header, for values that may be modified during iterations.
  These assignments select an initial value when the loop is entered at first and a loop computed value for each additional iteration.
- $\boldsymbol{\eta}$: at loop exit nodes, passing values calculated inside the loop out.
  These instructions make, in addition to ordinary SSA form, explicit which values are produced by a loop.

With TGSA form the control dependence of the loop header on the exit conditions is not considered in the μ nodes placed in the loop header. I.e., μ nodes do not take into account if the loop is terminated. In fact the μ assignments do not contain any predicate at all. Instead any value that leaves the loop is passed to a η assignment at the loop-exit nodes. The η node has the loop termination predicate as its sole predicate, describing which value is produced within the loop when the loop is terminated. This predicate is the inverse of the predicate required for execution of the loop back edge. It is the root of a DAG of γ assignments controlling the number of loop iterations. When at least one of the conditions in this predicate DAG is input dependent, the loop is considered input dependent.

4.2.3 Iteration Bounds

To guarantee that the control flow of the transformed program has the same execution trace regardless of input data, the loop-iteration counts have to be the same amongst executions. Therefore, for loops that have been identified as having their iteration count depending on input data by the dataflow analysis as described in Section 4.2.2, an iteration bound is required. This iteration bound, sometimes also called loop bound, must not depend on input data. Also it must not be lower than the iteration count of the loop would be, for any valid input data. This iteration bound is used by the SP-transformation to iterate the loop. Details about this part of the transformation can

[3] Strongly Connected Region

be found in Section 4.2.4. In the following paragraphs two approaches to determine loop bounds are described.

Annotations

The current implementation of the SP-transformation relies solely on source code annotations that specify the iteration bounds. Details about these annotations can be found in Section 3.3.2.

Automatic deduction

For certain programs it is possible to automatically deduce loop bounds. The current implementation does not implement nor use automatic loop bound deduction.

4.2.4 Transformation

This section describes how the current implementation of the SP-transformation transforms loops.

Transformation Scope

The scope of the transformation of a single loop starts at the loop header and ends at the basic block at which the control flow rejoins all edges leaving the loop. That means that the transformation is applied to a sub portion of the CFG dominated by the loop header and post dominated by the control-flow join-block. This clearly may contain some nodes that are not part of the loop itself. The handling of these nodes is described later on. The existence of such a control-flow joining node can be guaranteed after running the *Unify function exit nodes* pass as provided by LLVM, which guarantees that there is a unique basic block terminating the function. At least at this unique exit block, the control flow will rejoin.

The current implementation of the transformation requires that all control-flow paths originating from a certain loop, i.e., all paths starting with one of the loop's exiting edges, rejoin at a single basic block. Within reducible flow graphs it is however possible, that not all control-flow paths join at the same basic-block when there are more than 2 different paths exiting the loop. An example for this type of control flow is given in Figure 4.1.

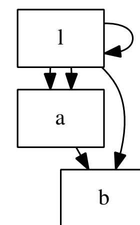


Figure 4.1: Unsupported Control Flow

The current implementation does not support this type of control-flow layout. Instead it requires that all control-flow paths started by the exiting edges of one loop, rejoin at a single basic block. Handling this kind of control flow requires the introduction of an additional condition, that would be set to true when the control follows a control-flow path that joins early. This condition would then be used to control the

execution of the basic blocks these code paths have in common, up to the point where the remaining control-flow paths join.

To find the node where the control flow rejoins, the post-dominance relation is used. Starting from the loop header the first post dominator, which is not part of the loop, is searched for, by walking the chain of immediate post dominators.

C Break Statements This paragraph shows why the restriction that all control paths leaving a loop have to rejoin at the same block still allows the transformation of a large portion of programs written in the C programming language.

The basic blocks, which lie outside of the loop, are usually generated by the usage of break statement within the C programming language. These basic blocks contain code for the instructions that immediately precede the break statement up to any control statement allowing a further loop iteration. A loop can contain several break statements. Each may post dominate several instructions in one or more basic blocks that are not part of the loop. But for every execution of the loop at most one of the break statements can be executed and, given the structured nature of the C programming language, each of them has independent code as long as one does not add goto instructions branching into the code preceding the break statements. As a consequence the C-compiler has no reason to create branches between the different loop-exiting paths.

Standalone Transformation of Loops The nodes dominated by the loop header and post dominated by the control-flow joining node form the scope on which the loop transformation works.

The nodes in a flow graph can be split into disjoint regions. An algorithm therefor is the interval construction as described in [6], [1] and [18]. To simplify program transformation it is, for certain transformations, possible to not transform the entire program at once but apply transformations to the individual intervals and merge the transformation results.

When a CFG is reducible, performing a reducibility test based on intervals as described in [6] page 442ff and [1], at some point a maximal interval will be created with the loop header being the interval header that contains at least all these basic blocks, either directly or as part of already reduced nodes. The interval must contain all nodes belonging to the loop started by the interval header when the flow graph is reducible, since the loop header dominates all blocks in a loop and an interval contains all nodes dominated by the interval header. Also the loop started by the interval header is the only loop that may be directly included in the interval, since other loops would not be included in interval construction, but start their own intervals. Other loops may of course be part of previously reduced intervals. Additionally the interval may contain some nodes outside of the currently handled loop, which do not need any processing when transforming the loop.

The interval based reducibility test is, at the outer level, merging graph nodes into intervals. Further applications of the interval construction merge graph nodes and previously constructed intervals. Every loop in the CFG results in the creation of its own

interval. So, in reducible CFGs the transformation can handle every loop without ever having to consider more than one loop at a time. Therefore it is required to handle already transformed intervals where the reducibility test would include an already reduced interval. The transformation could have been applied to the included interval before. Basic blocks in an interval that do not belong to the loop started by the interval header, are handled as described in the previous paragraph about *C Break Statements*.

Transformation Overview

In this section the effects of an application of the loop transformation are presented exemplarily. The exact steps taken in the course of this transformation are described in the following subsections. The transformations are shown based on the SSA form, since the implementation works with the LLVM IR which is in SSA form.

Figure 4.2 shows the scheme used by the current implementation to transform loops into SP-code. Figure 4.2a shows the loop before the transformation is applied. This example assumes, that this loop has previously been identified as input dependent by the dataflow analysis as described in Section 4.2.2.

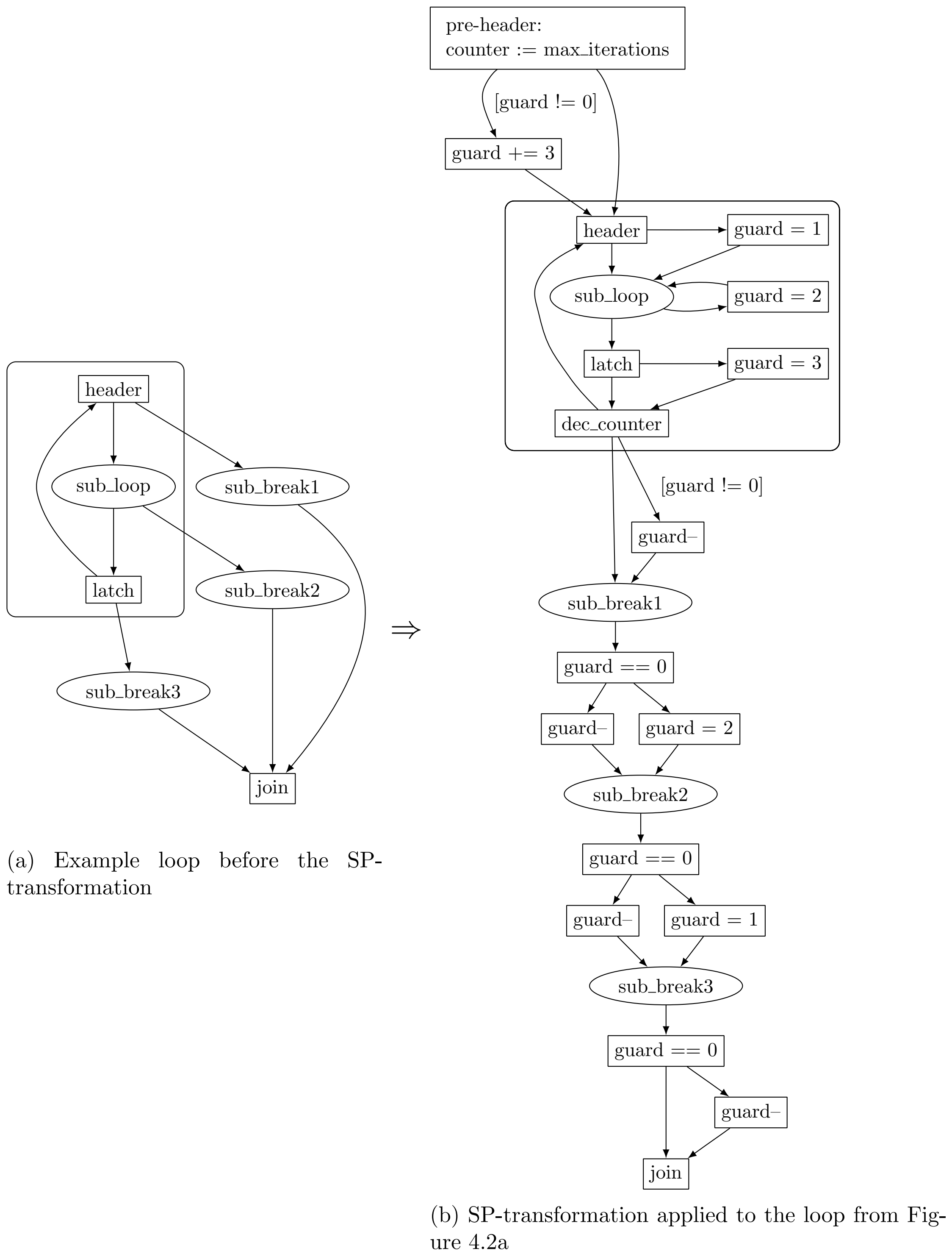
The entire graph shown in Figure 4.2a will usually be a subgraph of some larger CFG. These outer CFG parts are omitted in this example, because anything outside does not influence the transformation. Likewise do the elliptic nodes in Figure 4.2a denote entire subgraphs that may also contain further loops. These subgraphs may be independently transformed. The rectangular graph nodes represent, as usual, single basic blocks.

The natural loop shown in Figure 4.2a consists of the loop-header block *header*, the loop's latch block *latch* and the back edge *latch* → *header*. The remainder of the loop is shown as *loop_body*, a subgraph whose actual structure is not important to the loop transformation. To clarify which parts of the CFG belong to the loop, these parts are surrounded by an additional box.

Program parts that are transformed with the loop transformation, but are not part of the natural loop induced by the back edge *latch* → *header*, are the subgraphs *sub_break*1, *sub_break*2 and *sub_break*3. Their names reflect the fact that in the C-programming language, code immediately before a *break* statement is compiled into such CFG structures. The edges ending at the subgraphs *sub_break*1, *sub_break*2 and *sub_break*3 are the exiting edges of the loop in this example. Figure 4.2a depicts that the transformation has to expect exiting edges from each part of the loop. Additionally to the edges shown in this example, each of them may originate several loop-exiting edges. How the loop transformation handles this type of control flow is shown in detail in the previous paragraph *C Break Statements*.

The conversion is done in several steps which are described in the following sections. Before any transformation is carried out, the loop is prepared so that the CFG meets the general expectations of the subsequent transformation steps.

The description of these preparations is separated in several parts which follow in the next paragraphs. At first the introduction of an input-data independent loop counter is shown. Then the removal of the loop-exiting edges, which caused the original loop to



(a) Example loop before the SP-transformation

(b) SP-transformation applied to the loop from Figure 4.2a

Figure 4.2: Transformation of natural loops

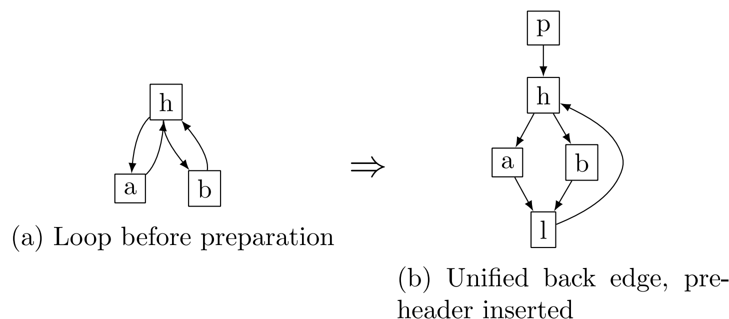
be considered input-data dependent, is explained in detail. Finally the transformation procedure required to create valid phi instructions for the modified CFG is described.

Preparing the Loop

This section describes the preparing steps that are required prior to applying the transformations shown in the subsequent sections. Transformations that should be avoided before the loop transformation are shown in the following paragraph *Notes on Loop Preparation*.

Pre-header A loop pre-header block with a single outgoing edge to the loop header is required. The pre-header block is the only predecessor of the loop header that is not part of the loop, i.e., the pre-header has to be the only block entering the loop. If such a block is already part of the CFG, the CFG remains unmodified. If no such block exists a new block is created, as done with node p in Figure 4.3b. The pre-header block is used by the loop transformation to add additional initialization code that needs to be executed once before the loop is entered.

Back edge The loop is required to have exactly one back edge after preparation. A single-entry loop may have several back edges targeting the same header block. That means, there are several natural loops sharing the same header node. It is always possible to condense them into a single natural loop by introducing a new latch node, re-target all back edges to this new latch node and introduce a new back edge from the latch node to the loop header. An example for such a new latch node is node l in Figure 4.3b.



(a) Loop before preparation

(b) Unified back edge, pre-header inserted

Figure 4.3: Preparation prior to loop transformation

Introduction of an Input-Independent Loop Counter

When the number of iterations of a loop depends on input data, it is rewritten in a way that the loop iteration count is no longer input dependent.

The requirement for an input-data independent iteration count with SP-programs is described earlier in Section 4.2.3 in the paragraph *Iteration Bounds*. The exact type of transformation that may be applied to loops depends on the way the iteration bound is

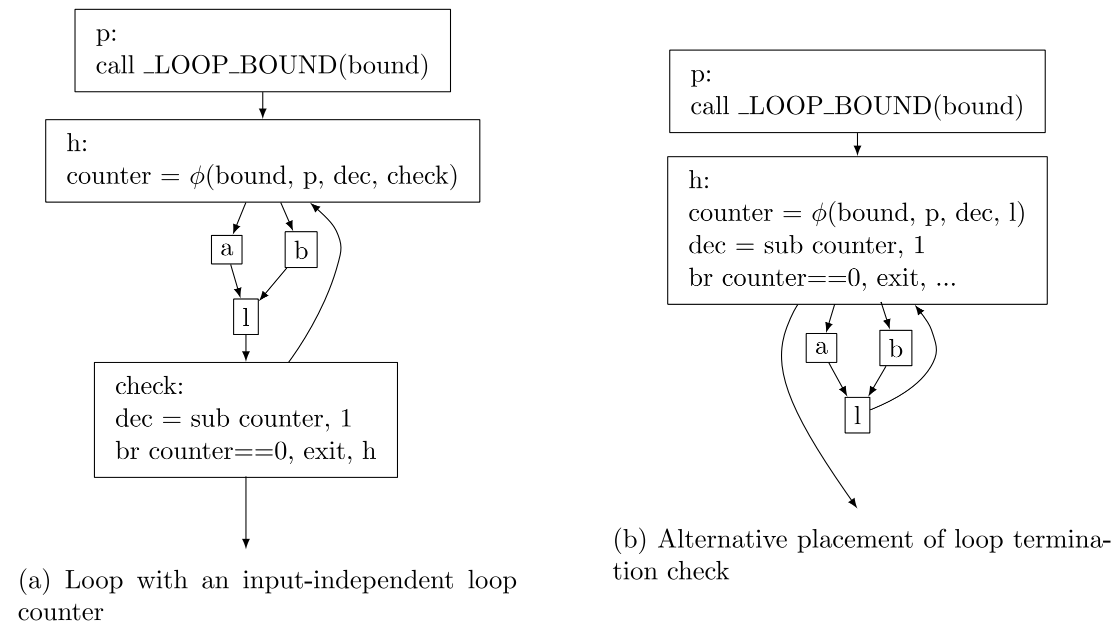
specified. In the current implementation the only supported way to provide loop bounds is through annotations as described in Section 3.3.2. The transformation described here requires exactly this kind of loop-bound annotation.

Further, the loop is assumed to have been prepared as previously described in the paragraph *Preparing the Loop*. The loop bound provided by the annotation is required to evaluate to a positive, input-data independent, integer value.

Figure 4.4a shows the effect to the CFG when applying the loop transformation as currently implemented to the loop from Figure 4.3b.

In the pre-header block the parameter to $_LOOP_BOUND$ is evaluated and assigned to the SSA variable bound. The loop-header block is prefixed by a new ϕ-instruction that is initialized with the value of the bound variable and decremented by one on each loop iteration. This ϕ-instruction is the new, input-data independent, loop counter determining the number of loop iterations. The sole loop-back edge is split to integrate a new basic block, called *check*, into each loop cycle. Within this block the value of the loop counter is checked and, as long it has not reached zero, the control flow branches to the loop header. If the loop counter has reached zero, the loop is left. The exact block to branch in this case is determined as described in paragraph *Handling Loop Termination*.

Figure 4.4b shows an alternative transformation scheme for loops, which differs from the currently implemented one by checking the loop counter before the first iteration. Applying this modified transformation would be beneficial for loops which are known to occasionally not iterate at all.



(a) Loop with an input-independent loop counter

(b) Alternative placement of loop termination check

Figure 4.4: Example loops with input-independent iteration count

Handling Loop Termination

A loop is considered input dependent when at least one of the branches exiting the loop is input dependent. Any input dependent branch has to be eliminated to guarantee a unique execution trace for each program execution. This section shows how the loop-exiting branches are removed in the current implementation of the SP-transformation.

When a loop-exiting edge is removed a guard setting block is inserted into the loop instead. This guard-setting block sets the guard value when the loop would have been terminated in the original program. The set guard value is chosen to reflect which of the loop-exiting edges would have been executed to leave the loop. Later on in the transformed program this new guard value determines the break-code fragment that should be executed after the loop has been left.

If the new guard setting block is control dependent on some branches contained in the same loop, these branches are subject to transformation by the subsequently executed SP-branch transformation. Since the SP-branch transformation also modifies the guard value, it has to take care that the value applied to the guard is incremented by 1 for every nesting level transformed within the loop.

The SP-branch transformation will later insert a pair of guard-increment and guard-decrement instructions within the loop. Whereby the guard-increment instruction will dominate the guard-setting instruction introduced to replace the loop-exiting edge. The guard-decrement instruction will post dominate the loop termination guard setter.

In the final active loop iteration at first the guard-increment instruction is passed by the control flow. But the predicates applied to this instruction will prevent it from being executed. Then, as a replacement for the loop-exiting edge, the guard value is set by the loop-exiting guard-setter. Before control flow reaches the loop latch it will pass the guard-decrement instruction. Now, since the guard value now reflects that the loop is terminated, the predicates applied to the decrement instruction will allow its execution. This decrement of the guard value has to be considered when choosing the guard value to reflect the loop-exiting edge that would have terminated the loop in the original program.

To communicate to the SP-branch transformation which guard values need to compensate subsequent guard-decrement instructions these guard-setting instructions are tagged by the metadata-tag *!inc_with_nesting_level*. When a SESE-region is transformed by the SP-branch transformation, as described in Section 4.3.5, all tagged instructions contained within a guard-increment/-decrement pair will have their guard values incremented.

Loop Properties Since the current implementation of the SP-transformation is only applicable to reducible flow graphs, all loops are expected to be single-entry loops. Further, each loop is expected to have been prepared as previously described. After these preparations it can safely be assume that only natural loops exist and that no two of the natural loops share the same loop header.

Following the definition for natural loops from [16], page 374, a natural loop defined by a back edge $l \rightarrow h$ is a triple $L = (N_L, E_L, h)$. With h being the loop header, l

being the loop latch and the set of nodes N_L, so that there is a path from each node $n \in N_L$ to the latch node l without passing the loop header h. Given the entire flow graph $G = (N, E, n_0)$, the set of nodes in the natural loop L is a subset $N_L \subseteq N$ of all the nodes in G. The set of Edges $E_L = E \cap (N_L \times N_L)$ in the loop is the subset of edges of the entire graph $E_L \subseteq E$, where source- and target-node are in N_L. Since in a reducible flow graph a loop may only be entered through the loop header, the loop header h dominates all nodes in N_L.

Exit Edge An exit edge is an edge that leaves a loop. An exit edge originates from any of the nodes in N_L and ends at a node not part of the loop, i.e., a node in $N \setminus N_L$. The set of exit edges for a loop L is $E_{exit} = \{\langle u \to w \rangle \in E | u \in N_L \wedge w \in N \setminus N_L\}$. Following the definitions from the previous paragraph, when the control flow follows any of the edges in E_{exit} there may be no further iteration of the loop's back edge without re-entering the loop through the loop header again.

The conditions within a loop, that control the execution of the loop-exit edges, form an SCR in the CDG[4]. Since input dependence is propagated by control dependencies, whenever one of the branches in the SCR is control dependent, all other branches in the SCR are input dependent as well. As a consequence either all exiting branches of a loop are input dependent or none of them is input dependent.

When one exiting branch has been determined to be input-data dependent, all other exiting branches are control dependent on this branch and are therefore also considered input-data dependent. This is the reason why there is no need to distinguish between input-dependent and input-independent branches when handling the loop exits of a single loop during SP-transformation.

Removing the Exit Edges During loop transformation at first the exit edges are numbered. Numbering is done one based, so the edge numbers can be directly assigned to the guard value later on. A guard value of zero is reserved to represent program parts that should be executed.

When a loop is terminated in the original program, exactly one of its exiting edges is executed. The transformed program removes the exiting edge and instead sets the guard value to the exiting edge's number. The exiting edge's number is also used as an execution condition for the basic block that is targeted by the exit edge in the original program. As noted earlier, the current implementation expects that the code paths starting with loop-exit edges have no nodes in common until the control flow of all these paths join again. So each path leaving a loop has exactly one such condition assigned that is valid until all paths join.

Edge Numbering For each exit edge a new guard-setting block is created that sets the guard value to the number determined for the exit edge. The loop-exiting edge is re-targeted to the new block. As an example see block g in Figure 4.5b.

[4]Control-Dependence Graph

The numbering of the exit edges and using these numbers as execution conditions is closely related to the transformation of exit branches as described in the original publication on IF conversion in [3] on page 180 ff. They introduced for each exit branch an exit flag that would evaluate to true when an exit branch would not yet have been executed. The exit branches are replaced by an assignment of the negated predicate-evaluation result that is required for the exit branch to be executed. All the exit flags are then conjuncted and assigned as a predicate to all statements within the loop including the exit flag assignments. As a result at most one of the exit flags, the one representing the exit branch that would have been executed first, will have a value of false assigned. The value assigned to the guard as described in the above paragraphs may be thought of representing the number of the exit flag that would have assigned a value of false in the original IF conversion.

Guard-Setter Placement The new guard-setting block is integrated into the loop's control flow with an unconditional branch to another node that is part of the loop. It logically terminates the loop by causing further evaluations of execution predicates for basic blocks within the loop to yield false. In the CFG this corresponds to the creation of an unconditional control-flow edge from the guard-setting block to the successor block. The successor block for the new guard-setting block is determined by choosing an arbitrary successor of the loop-exiting block that is part of the loop. The current implementation uses the first one found. Every loop-exiting block must have at least one such successor since the exiting blocks are, by definition, part of the loop. When the target block contains phi statements, the new incoming edge from the set-guard block is added to the phi statements. The phi's value for this new incoming edge is set to be the same as for the incoming edge from the exiting block, since the guard-setting block should not modify anything but the guard value. The phi instructions in the loop-exit nodes require some additional handling that is shown in Section 4.2.4.

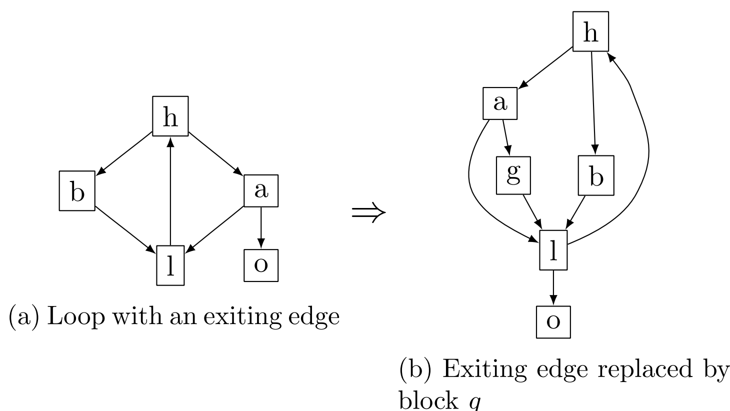
The only outgoing edge of each guard-setting block targets a node that is part of the loop. This is a node that must have been on a path to the loop latch without passing the loop header before. Such a successor node exists, by definition, for any node that is part of a loop, with the exception of the loop latch node. The selection of a successor node for the loop latch is even simpler, since it can always branch to the loop header. That means, that the guard-setting block is guaranteed to be part of the loop and its predecessors and successor are also part of the loop. When all exiting edges are replaced by guard-setting blocks, none of the remaining edges is exiting the loop now. A new exiting branch has to be created with the introduction of an input-independent loop counter.

Reducibility The incoming edge to the new guard-setting block does originate from a basic block that is part of the loop. The originating block of this edge was originating a loop-exiting edge in the original program and each block originating a loop-exiting edge is contained in the loop. Since the transformation only works on reducible flow graphs that means that the loop header dominates the block that originated the loop-

exiting edge. The new guard-setting block has this block as its only predecessor, so it is also dominated by the loop header. Further the branch originating the edge from the guard-setting block to its successor block is one of the branches that existed before the transformation. But the edge is now split with the guard-setting block as its intermediate block. As a result the graph is guaranteed to remain reducible after this transformation.

When this transformation is applied to a loop, the loop-exit branches are all input-dependent as shown in Section 4.2.2. This fact is communicated across transformation passes by the use of the *!IDD* metadata tag. The former exit branches, which are now redirected to the new guard-setting blocks, keep this metadata flag that marks them as input-data dependent to ensure further transformation by the SP-branch transformation pass later on.

Example The example in Figure 4.5a shows a loop $L = \{h, a, b, l\}$ with an exiting edge $a \rightarrow o$. The transformed CFG shown in Figure 4.5b has the loop-exiting edge replaced by the new basic block g. This block is expected to contain code that takes care of the logical loop termination. Since the previous exiting edge has been removed, a new exiting edge $l \rightarrow o$ is introduced that allows exiting the loop when the input-independent number of loop iterations has been performed.



(a) Loop with an exiting edge

(b) Exiting edge replaced by block g

Figure 4.5: Replacing loop exiting edges

Rewrite ϕ-Instructions in Loop-Exit Blocks

In a CFG, loop-exit blocks are the basic blocks outside a loop that have predecessors inside a loop. Exiting blocks are the basic blocks within the loop that branch to successors outside the loop. The control-flow edges from exiting blocks to exit blocks are called loop-exit edges. A loop is terminated when the control flow follows one of the loop-exit edges since there is no path from a loop-exit node to the loop's back edge without re-entering the loop through the loop header. That guarantees that, whenever a loop is entered through the loop header at most one of its exiting edges will be executed before the loop is entered through the loop header again. Note that in general a loop may also be left by an exiting edge originating from some nested loop or by execution

of a return statement leaving the entire function, but the current implementation of the SP-transformation does not support these types of control flows.

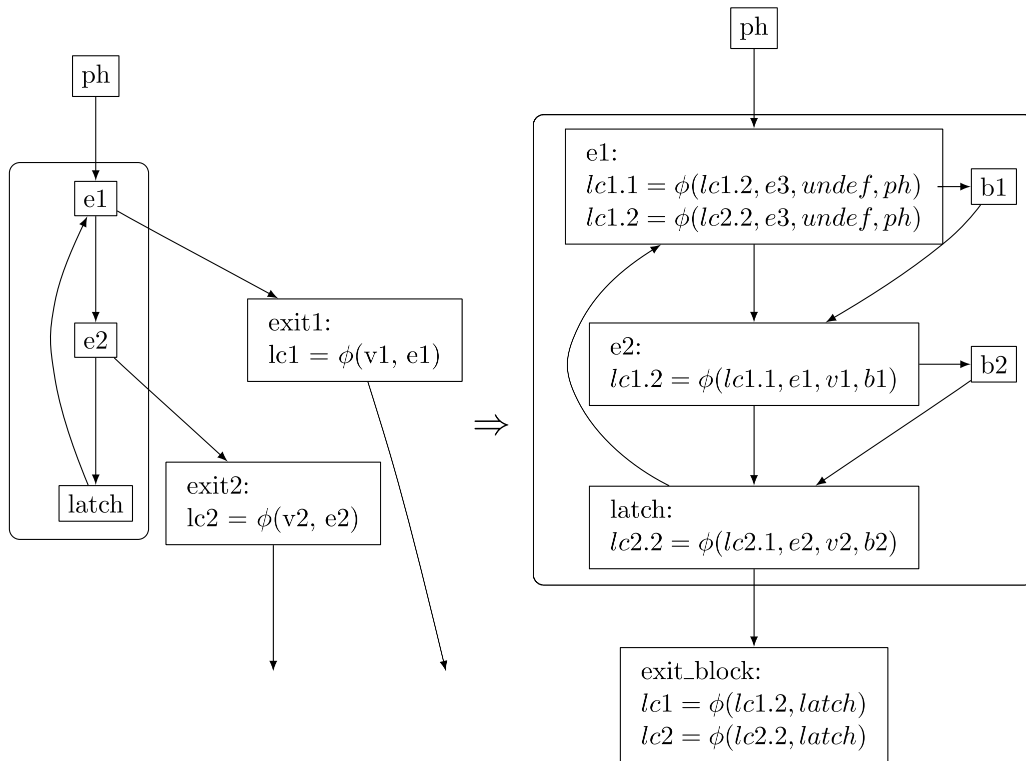
When rewriting the loop exits the SP-transformation distinguishes two types of ϕ-instructions by the origin of their incoming values. They may be either calculated within the loop that is exited, or not. In the former case, the ϕ-instruction has only a single incoming value that is calculated within the loop. In the later case the ϕ-instruction is selecting amongst loop-invariant values. A different transformation is applied to the ϕ-instruction in each case. These transformations are shown in the following paragraphs.

Loop Calculated Values The loop's exit nodes, those are the nodes that have been targeted by at least one of the loop-exiting edges may, after the previous transformation step, be unreachable from other nodes in the CFG since the exiting edges have been re-targeted at the guard-setting block as described above in the paragraph *Removing the Exit Edges*. As already stated, control flows with other incoming edges to the exiting blocks are not supported by this transformation. The following describes the steps taken to re-integrate the exit blocks into the control flow.

LLVM provides the *Canonicalize natural loops* pass that guarantees, in absence of indirect branches, that all the exit blocks only have exiting blocks as their predecessors. Executing this pass before starting the loop transformation guarantees that the former exiting blocks have no incoming edges after the previous transformation step since this transformation step removed all the exiting edges.

Another prerequisite for this transformation is that the loop is required to be in LCSSA[5]-form, therefore any values produced by the loop are made explicit through ϕ-instructions in the loop-exit blocks. The example provided in Figure 4.6a has the ϕ-instructions *lc1* and *lc2* renaming the loop defined values *v1* and *v2*. During the transformation these ϕ-statements are split up into several ϕs which are placed inside the loop, selecting a value depending on the execution of the guard-setting blocks *b1* and *b2* that have previously been introduced to replace the loop-exiting edges. For an example of this type of ϕ-instructions see *lc1.2* and *lc2.2* in Figure 4.6b. Subsequent branch transformation, as described in Section 4.3.5, makes sure that exactly the value is selected by the ϕ-instruction that would have been selected by that loop exit which would have been executed in the original program.

[5]Loop-Closed Static Single Assignment (form)



(a) ϕ-instruction in the loop-exit block

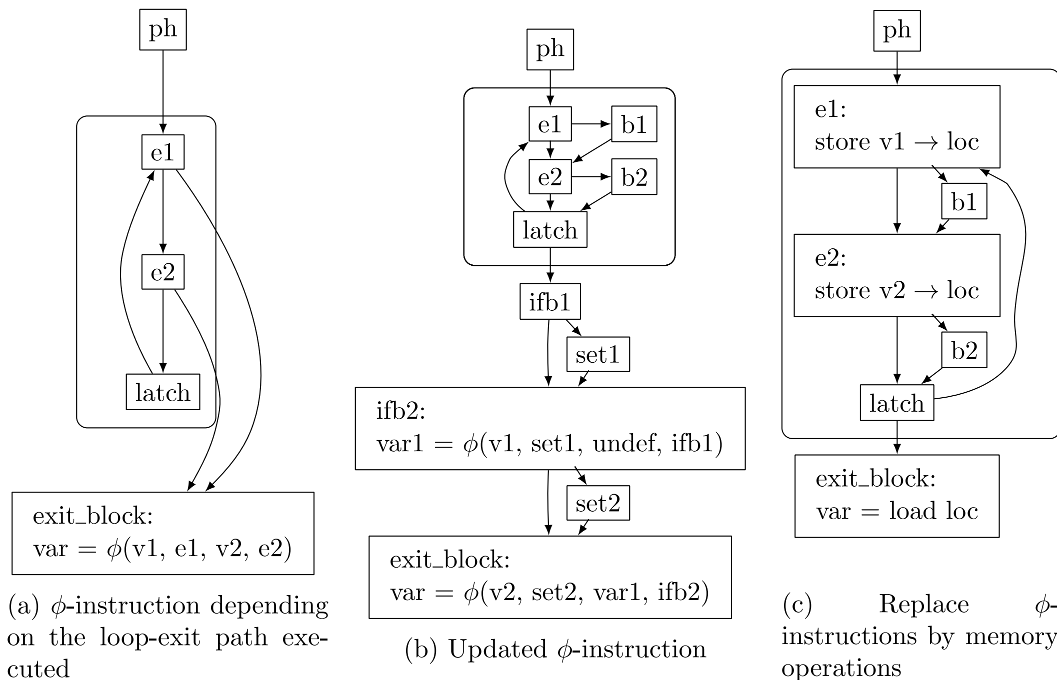
(b) Updated ϕ-instruction

Figure 4.6: Rewriting ϕ-instructions of loop exits

ϕ-Instructions with Loop-Invariant Values The code paths leaving a loop may contain ϕ-statements that choose values depending on the path the loop is left on, but those incoming values are not generated within the loop. The SP-transformed loop is exited on one unique code path. So selecting a value depending on the executed code path is no longer feasible. Two possible transformations for these ϕ-instructions are described in the following paragraphs *Update ϕ* and *Demote all Values to Stack Slots*.

Update ϕ The example in Figure 4.7a shows a CFG with a ϕ-instruction selecting values depending on the executed loop-exit edge. The values selected are considered to be loop invariant. Otherwise an additional loop-exit block copying the value would be part of the CFG since the loops are expected to be in LCSSA-form. Figure 4.7b shows how these values could be handled keeping their SSA-form. In cases where additional basic blocks on the exit edges exist and any one of the incoming values is defined in these basic blocks the transformation would have to place the blocks containing the definitions so that they dominate the ϕ-instructions with their uses to keep the program in valid SSA-form.

Demote all Values to Stack Slots The current implementation uses the approach shown in Figure 4.7c since it is easier to implement than the transformation shown in the above paragraph *Update* ϕ. The ϕ-instructions are replaced by load and store operations targeting stack memory locations by applying the *Demote all values to stack slots* transformation that is provided by LLVM. The load and store operations are guarded in a later transformation step, not shown in the example graphs, to keep the program's semantic. Additionally, after guarding but before the register allocation is performed, the memory operations can be transformed back into SSA registers to prevent the performance impact these memory accesses would impose.



(a) ϕ-instruction depending on the loop-exit path executed

(b) Updated ϕ-instruction

(c) Replace ϕ-instructions by memory operations

Figure 4.7: Rewriting ϕ-instructions on different loop-exit paths

Notes on Loop Preparation

LLVM provides several transformation passes that may be used to ensure that the program can be transformed by the SP-transformation implementation.

The first transformation that should be executed is the *Canonicalize natural loops* pass. This pass ensures that all loops have exactly one back edge which prevents further transformation passes from splitting up the loop into several nested loops.

The remainder of this section shows transformations that should be avoided before applying the SP-transformation described in this document. These transformations may modify the loops in a way that they may increase the programs runtime by a large amount. The following paragraphs illustrate the negative effect of certain transformations by showing the results of their application to the program given in Listing 4.1.

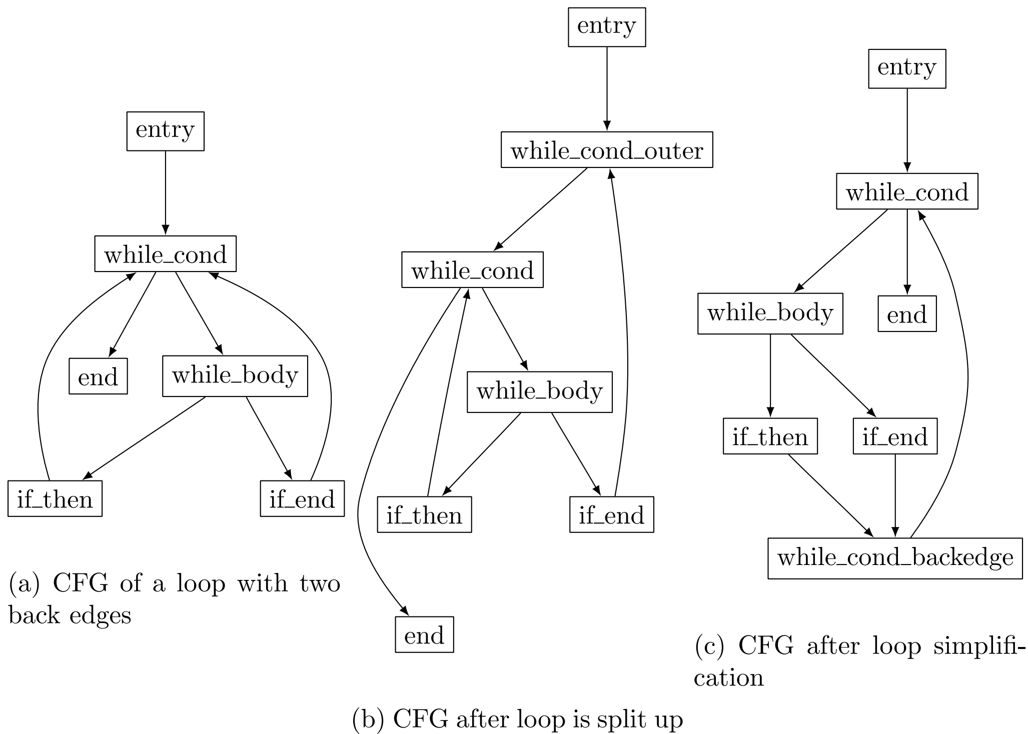
```
int x() {
    int y,z;
    while(--y) {
        if(y % 4)
            continue;
        ++z;
    }
    return z;
}
```

Listing 4.1: Loop with two back edges

Compiling the code from Listing 4.1 using *clang* without any optimizations yields the CFG shown in Figure 4.8a. This CFG contains two back edges. These are the edges from *if_then* $\rightarrow$ *while_cond* and *if_end* $\rightarrow$ *while_cond*. These two back edges form two natural loops $L_1 = \{$ *while_cond, while_body, if_then* $\}$ and $L_2 = \{$ *while_cond, while_body, if_end* $\}$.

The application of the transformations *Promote Memory to Register* and *Canonicalize Induction Variables* to the CFG in Figure 4.8a yields the CFG shown in Figure 4.8b. The CFG still contains two back edges but they have differing target nodes now. They form two nested natural loops with differing loop headers. $L_1 = \{$ *while_cond, while_body, if_then* $\}$ and $L_2 = \{$ *while_cond_outer, while_cond, while_body, if_then, if_end* $\}$. When the loop L_1 is subsequently modified to have an input-independent iteration count, any execution-time overhead caused by this transformation is multiplied by the iteration count of L_2. Additionally, from here on calls to loop simplify are no longer able to combine the two back edges into one and create a single natural loop.

Applying the *Canonicalize natural loops* transformation first transforms the CFG from Figure 4.8a into the CFG shown in Figure 4.8c. Note that this transformation is possible for all natural loops sharing the same header node by introducing a new loop latch, called *while_cond_backedge* in this example, and redirecting the existing back edges to the new loop latch. The loop has now only one back edge and a unique loop header. Subsequent calls of the *Promote Memory to Register* and *Canonicalize Induction Variables* transformations do not cause modifications to this CFG.



(a) CFG of a loop with two back edges

(b) CFG after loop is split up

(c) CFG after loop simplification

Figure 4.8: Example CFGs showing how a wrong application order of loop optimizations could degrade execution performance

Do not Run the *Simplify the CFG* Pass after *Canonicalize Natural Loops*
The *canonicalize natural loops* pass ensures that every loop has exactly one back edge. It does so by inserting an additional, empty, loop latch block if necessary. The *Simplify the CFG* pass should not be run after the *canonicalize natural loops* pass, because *Simplify the CFG* will remove such empty blocks, yielding two back edges for the loop again.

Do not Run *Loop Invariant Code Motion* before *Canonicalize Natural Loops*
When a loop has more than one back edge the *Loop Invariant Code Motion* pass may possibly split the loop header, thus creating additional nested loops for some, or each, of the back edges. Whereby inner loops may contain exiting edges which do not only leave the inner loop but also one or more of the containing loops. It may be possible to transform the resulting CFG into single-path code but the current implementation of the single-path transformation does not support this type of control flow. Further the single-path transformation may cause runtime growth that is polynomial to the number of nested loops generated when inexact iteration bounds are determined for the nested loops.

Removing Marker-Function Calls

Once the loop transformation has been executed, the *_LOOP_BOUND* annotation functions are no longer required. Also the data-dependency analysis pass is required to have already been executed before, since it is a prerequisite for the loop transformation pass. As a result the marker calls to the functions *_ID* and *_NID* are no longer required. The *SP Remove Markers* pass is provided to remove any calls to these functions. It is recommended to execute this pass immediately after the *SP loop transformation* pass, so that the marker calls do not interfere with further optimization passes.

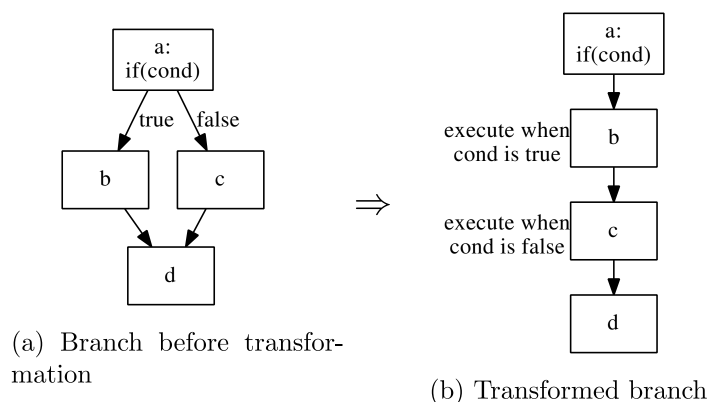
4.3 Branches

This section discusses the transformations steps applied to branches while transforming a program to single-path code. The branch transformations described here are done in a separate transformation pass called the *SP branch transformation* pass.

Transformation Goal The SP-transformation aims at creating a program that has a unique execution trace for each invocation. To achieve this the program is prepared so that, at each point in the program, at most one input-data dependent predicate controls the execution of program statements. This predicate can then be applied using the constant-time conditional expression [35] by a later transformation pass resulting in a program with a unique, constant execution time.

The foundation for the *SP branch transformation* pass is the forward branch transformation shown in [3].

The basic principle of the branch transformation is illustrated in Figure 4.9, whereby Subfigure 4.9a shows the CFG containing a branch that needs to be transformed. After application of the transformation the resulting CFG will look like the one shown in Subfigure 4.9b.



(a) Branch before transformation

(b) Transformed branch

Figure 4.9: Example illustrating the branch transformation

4.3.1 Notations

A short description of the common notations used in conjunction with control-flow graphs that are used within this section. Mostly from [2], [17] and [16], page 374.

Topological Order An order of all nodes in a DAG, so that any node n is preceded by all nodes on all possible paths from start to n.

Forward CFG In reducible control-flow graphs the back edges can be uniquely identified by establishing a DFST[6] over the graph and finding all CFG edges pointing to an ancestor in the DFST. The exact algorithm is described in [16] on page 371. The FCFG[7] of a flow graph is obtained by removing all back edges from the CFG.

Branch Types IF-conversion in [3] distinguishes 3 types of branches:

- **Exit branch** Branches leaving a loop. Branch origin and target differ on loop nesting level.
- **Forward branch** Branches at the same loop nesting level.
- **Backward branch** Branches causing irreducible control flow. Note that these are not the loop back edges.

4.3.2 Branches to Transform

The transformation described in this section is only applicable to branches that are not the origin of a loop-back edge. The term forward branch has been used for this type of branch in [3]. These are the control-flow branches and joins in the FCFG. The loop-back edges are handled as part of the loop transformation as described in Section 4.2. A previous run of the loop transformation is expected to have reduced the input-data dependent exit branches to forward branches before.

To enable the application of the *SP branch transformation* to forward branches only, foremost these branches have to be identified. The other types of branches identified in [3] are exit branches and backward branches. The following paragraphs will point out the properties that separate the forward branches from the other types of branches.

Forward Branches These are the kind of branches that the *SP branch transformation* strives to remove. These branches occur within the same loop nesting level, i.e., the basic block containing the branch instruction and all basic blocks that are targeted by this branch are contained within the same loop.

In the C programming language [47] they may be introduced by a conditional statement, i.e., *if*. Another common source for this type of branches in the C programming language are logical operators within conditional statements. E.g., the logical-and operator *&&* guarantees by specification ([47], page 89) that its second operand is not evaluated if the first one evaluates equal to 0. Clang, one of the C front-ends to LLVM, translates that into a conditional branch.

[6] Depth-First Spanning Tree

[7] Forward Control Flow Graph

Exit Branches The input-data dependent exit branches have already been removed by the *SP loop transformation*. The remaining exit branches may be ignored by the *SP branch transformation* since they have to be input-data independent.

Backward Branches

> *... branch to a statement occurring lexically before the branch but at the same nesting levels ...*
>
> ---
>
> Definition of backward branches, from [3], page 179

This classification was sensible since the if-conversion in [3] is entirely described on a FORTRAN source-code level. FORTRAN provides statements for iterative execution, the so called DO-formulas allowing the specification of iterative program parts. The implicit loops defined by backward branches were concerned separated from explicit loops by the original if-conversion.

Since the branch transformation implemented with the SP-transformation works on LLVM IR level using CFG analysis to identify loops there is no need to differentiate between branches introduced by loop statements and other backward branches. When the program has been transformed into LLVM IR, the way a loop has been specified in the source code is no longer reflected in the IR.

Part of the problem the original if-conversion had with backward branches was that they could introduce irreducible-control flow. The current implementation of the SP-transformation does not support irreducible-control flow. It requires its input programs to be reducible. Why irreducible-control flows are difficult for the SP-transformation is described in Section 2.3.

All branches in the FCFG, which have been identified as input-data dependent by the dataflow analysis, have to be removed by the transformation described here. The exact steps are described in the following sections. First Section 4.3.3 describes program preparation that are presumed by the subsequent transformation steps. Then Section 4.3.4 shows how execution predicates, that control the execution of program parts after the removal of branches, are determined. The procedure to remove the branches from the program is explained in Section 4.3.5. Finally Section 4.3.6 describes how the program is reordered after branch removal.

4.3.3 Preparation

To simplify the implementation of the branch transformation, the program is preprocessed to constrain the CFG as follows. Any CFG node has at most 2 successor nodes. The number of SESE-regions is maximized by allowing only 2 incoming edges for region-join nodes. In general these preparations require the introduction of additional basic blocks, branches and phi-instructions.

Terminator Instructions In the LLVM IR basic blocks always have a so-called terminator instruction as their last instruction. The language specification [33] for the LLVM IR contains the terminator instructions *ret*, *br*, *switch*, *indirectbr*, *invoke*, *resume* and *unreachable*. The current implementation of the branch transformation is limited to support only the terminator instructions *ret* and *br*. Programs that contain any of the other terminator instructions will lead to compilation errors. This constraint simplifies the control-flow analysis.

LLVM provides the *Lower SwitchInsts to branches* pass which converts switch instructions to an equivalent series of branch instructions. Whenever the program that should be transformed contains a switch instruction this transformation pass has to be executed before the branch transformation pass.

The remaining terminator instructions *indirectbr*, *invoke*, *resume* and *unreachable* have to be avoided within programs to which the branch transformation should be applied. This can be done by modifying the program that is transformed by using a different compiler frontend or by implementing another preprocessing transformation that removes them. Experiments in the course of testing the SP-transformation implementation have shown when compiling simple C sources with the clang compiler frontend the compilation does not make use of any of these terminator instructions.

As an additional restriction the *ret* instruction is only allowed once in each function, namely at the end of the function. Stock LLVM provides the *Unify function exit nodes* pass that ensures this property by adding a new basic block that only consists of a *ret* instruction. Additionally it creates branches from all the basic blocks that previously contained *ret* instructions to the new one. The old *ret* instructions are removed from the program. To select the return value that is passed as an argument to the *ret* instruction phi-instructions are introduced where required.

Single-Entry Single-Exit Regions CFGs may be decomposed into a tree, the PST[8], of SESE-regions. The PST simplifies the predicate determination as described in Section 4.3.4. SESE-regions have been defined to be bound by entry and exit edges [19] or entry and exit vertices [46]. SESE-regions connect at exactly two nodes or vertices, called region entry and region exit, to the remainder of the graph.

The *SESE-region identification* pass provided by LLVM locates SESE-regions bound by vertices. The current implementation of the SP-transformation uses the analysis results of this pass. Subsequent references to SESE-regions will refer to vertex-bound regions unless otherwise stated.

LLVM's SESE-region detection distinguishes canonical and complex regions. SESE-regions that cannot be split into two disjoint regions are called canonical SESE-regions. All manipulations described below work on canonical SESE-regions unless otherwise noted. For an exact definition of canonical SESE-regions see [19] on page 172. Complex regions may be split into two or more canonical SESE-regions by aggregating entry or exit edges into new CFG nodes. Note that a complex region is not necessarily a SESE-region until this region splitting has been performed.

[8]Process Structure Tree

The branch transformation transforms SESE-regions independently from each other as shown in the remainder of this section. To simplify this separate transformation of SESE-regions as a preprocessing step, any complex regions in the program are transformed into canonical regions so that no two regions share their entry or exit nodes. This way, whenever modifications to a phi instruction are required all incoming edges to the basic block containing the phi-instruction originate from the same SESE-regions. So the phi rewriting algorithm does not have to be able to handle incoming edges from other SESE-regions than the currently transformed one.

Splitting Complex Regions Since the supported terminator instructions are limited to *br* and *ret* as described above, no basic block can have more than two successors. That already ensures that no two regions share the same entry block.

When regions share a common exit block, starting from the innermost region a SESE-region is picked and a new exit block is created for this region. The new exit block branches to the original exit block of the complex region. All exiting edges belonging to the picked region are redirected to the new exit block. Phi-instructions are updated as required. This is repeated until each SESE-region has its unique exit block. When completed, no more complex regions exist.

4.3.4 Predicate Determination

Since the branch transformation works on forward branches only, which are by definition oriented in a forward direction w.r.t. the program flow, one can distinguish branch and join locations in the CFG. Note that the unambiguously determination of a forward direction in the program requires the control flow to be reducible.

Control-flow branches are nodes in the FCFG with more than one outgoing edge. Control-flow joins are nodes with more than two incoming edges in the FCFG.

The predicate combination described here corresponds to the one described in [3] on page 182f.

A basic example how the execution predicate that is calculated for the individual basic blocks is modified during forward-branch removal is given in Figure 4.10. This example contains a single branch with two branch targets. The execution of these branch targets depends on the value determined by the evaluation of *cond*. When *cond* evaluates to *true* the program's execution would continue at basic block a, should it evaluate to false the program's control would be transferred to block b. Both blocks, a and b, have a single successor c which is executed subsequently. The execution of node c does not depend on the evaluation result of *cond*. Note that, since the CFG given in this example contains no loops, FCFG and CFG are the same.

The guarding conditions that are determined during forward-branch removal for the graph in Figure 4.10 are printed on the right side of the CFG nodes. The condition for the entire CFG to be executed is denoted by *cc* which corresponds to the current condition of the example in [3] on page 182f.

Edge Predicates

Since the SP-transformation is implemented as a control-flow analysis based approach it is natural to think of execution predicates attached to the edges in the CFG. In the following these execution predicates are called edge predicates.

Edges in a CFG represent the transfer of control from one basic block to another. An edge predicate is expected to evaluate to true iff the control is passed from the originating basic block of the edge to the target block of the edge during program execution. Since, for all edges outgoing from a single basic block, control can only be transferred along exactly one edge, the edge predicates of all outgoing edges for any basic block have to be disjoint.

Two conditions have to hold for the program control being transferred between two basic blocks adjacent to an edge in the CFG:

1. The block originating the edge must currently have program control. In this case the execution condition determined for this block must evaluate to true.
2. When the originating block has multiple outgoing edges, the edge under consideration must be chosen to transfer control. This is when the evaluation of the branch condition matches the edge predicate attached. For blocks with only one outgoing edge this is always true.

As an example, Table 4.1 lists the edge conditions for the CFG given in Figure 4.10.

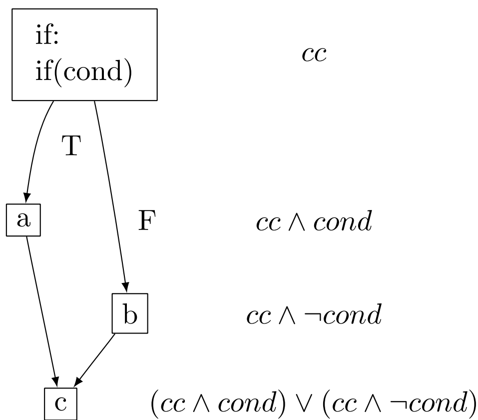


Figure 4.10: Mapping of control-flow branches and joins to guarding conditions

Edge	Edge condition
if→a	$cc \wedge cond$
if→b	$cc \wedge \neg cond$
a→c	$(cc \wedge cond) \wedge \top$
b→c	$(cc \wedge \neg cond) \wedge \top$

Table 4.1: Edge conditions for the CFG in Figure 4.10

Control-Flow Branches

The branch condition that selects the edge used to transfer the control at control-flow branches clearly has to be part of the execution predicates of their successing basic blocks. Since when the basic block containing the branch is executed, the branch condition determines the outgoing edge selected to transfer control. A notable exception is the case where the branching block dominates the successor block, i.e., multiple paths exist

from the branching block to the successor block. The current implementation of the SP-transformation does not handle this case specially. Although the determined execution conditions may be more complex than necessary, they will still be correct.

The determination of the execution conditions is implemented as a top-down process applied in topological order of the CFG. When a branch is encountered, the branch condition is attached to the outgoing CFG edges of the current block as described in Section 4.3.4. An execution condition for the successor blocks is later created out of these edge conditions. This process is described in the subsequent paragraphs about the handling of incoming CFG edges.

The example in Figure 4.10 illustrates the predicate calculation. It contains one branch in the basic block labeled if. Depending on the condition *cond* the program control is either transferred to basic block a or b. For any of these branch targets to be executed, 2 conditions have to hold. The current condition *cc* that is guarding the execution of the branching block must evaluate to *true* and the evaluation of the branch condition *cond* has to match the incoming edge's condition attached in the CFG. The conjunction of these conditions yields the execution conditions for the blocks a and b as it can be seen in Table 4.1.

When viewing the execution condition for basic block a that means *cc* has to evaluate to true and *cond* has to evaluate to true, what results in the guarding condition $cc \wedge cond$. Similarly for block b, but since this block is on the false side of the branch, denoted by the edge label F, the condition *cond* has to evaluate to false. The complete condition is therefore $cc \wedge \neg cond$.

Single Outgoing Edge Basic blocks that have only a single outgoing edge in the CFG are handled in a similar manner. Since whenever a basic block with a single outgoing edge is executed its single successor block will also be executed. In this case no additional conditions have to be regarded in the successor's execution predicate for this particular edge type. The edge condition of the single outgoing edge therefore equals the execution predicate of the originating basic block. This is exemplarily shown by the conditions for the edges $a \to c$ and $b \to c$ in Table 4.1.

Control-Flow Joins

In [3] the combination of predicates at branch-target locations in FORTRAN sources is described. At these code locations the control flows, which originate from the branch origin and from the preceding FORTRAN statement, merges.

The implementation of the SP-transformation is based on LLVM's IR, in which no fall-through from one basic block to another is specified [33]. Instead basic blocks may have multiple branches targeting them, or from a CFG's perspective have several incoming edges. The merging of predicates at control-flow joins described here is applied to these basic blocks.

Control flow joins are basic blocks in the FCFG with more than one incoming edge. The basic blocks from which these edges originate are called the predecessor blocks. Since an edge in the FCFG represents the transfer of control from the edges originating

basic block to the target block [2], a basic block with more than one incoming edge will gain control when any of its predecessors had control and transferred it through the incoming edge. The fact that a basic block with multiple outgoing will transfer control exactly along one of its outgoing edges is already encoded in the edge predicates of these outgoing edges, which are determined as described in the previous paragraphs. Therefore the predicate at control-flow joins can be determined by the disjunction of the edge predicates of all incoming edges.

In the example given in Figure 4.10 the only basic block joining the control flow is block c. The disjunction of the predicates calculated for the incoming edges $a \rightarrow c$ and $b \rightarrow c$ as given in Table 4.1 yields the predicate $(cc \wedge cond) \vee (cc \wedge \neg cond)$.

Single Incoming Edge Basic blocks with just one incoming edge are handled exactly like control-flow joins. Since there is just one incoming edge, the complete condition will equal to the predicate minterm induced by the edge predicate of the single incoming edge.

Simplification of Predicate Expressions

The execution predicates that where determined as described in the previous paragraphs may possibly be equivalent to simpler expressions. For example the condition for basic block c in the example given in Figure 4.10 is overly complicated since $cc \equiv (cc \wedge cond) \vee (cc \wedge \neg cond)$. The original publication on if-conversion [3] suggests the application of boolean simplifications based on the Quine-McCluskey prime implicant simplification [45][28].

The implementation of the SP-transformation does currently implement no simplification of the execution predicates. Since the ultimate goal of the entire transformation process described here is to generate a program that shows an input-data independent instruction trace when executed, any of the branches that where the reason for the generation of the execution-predicate expression should be replaced by an execution under control of the constant-time conditional expression.

Since the system targeted by the current implementation does not directly provide constant-time conditional expressions that could be used for optimized predicate calculation, conventional execution is used instead by the current implementation. To keep the SP-property of the transformed program the predicate calculation is implemented in a branch free block of combinatorial instructions. The individual combinatorial instructions execute in constant time on the target system.

Not having implemented execution-predicate simplification requires that the predicates for all incoming edges will have to be evaluated before executing any basic block. Since in the SP-program any blocks originating the incoming edges must have been executed before, the edge conditions for all incoming edges are available when control reaches the control-flow joining basic block. Anything left to determine the execution predicate for the joining basic block is the disjunction of the predicates of the incoming edges. The implementation stores the evaluation results of the execution predicates in

SSA variables, leaving the register allocator with the task of finding an efficient memory mapping for these values.

When composing SESE-regions which are independently transformed the implementation makes use of the fact that the predicate evaluation of the entry and exit block of a particular region must always yield the same result. A prerequisite therefore is that the entry and exit blocks are unique to one region, which in turn is guaranteed after applying the preparation processing described in Section 4.3.3.

Predicate Build Order

As described above the execution predicate for any basic block is a combination of execution predicates of its predecessor basic blocks and edge conditions when a predecessor block has more than one outgoing edge. The execution predicates for the individual basic blocks are built in topological order of the FCFG. This guarantees that the predicates of all blocks originating the incoming edges to a block had their predicates already determined.

The reverse postorder may be used to establish a topological order for any FCFG originating from a reducible CFG [17].

Predicates and the PST

The PST [19] is a tree representation of all SESE-regions in a program. LLVM's SESE-region identification pass has been modeled after the ideas in [19] and [46], as is stated in the source comments. The fragments organized in the PST as defined in [46] are sets of edges which are bound by an entry and exit node to the SESE-region.

Aggregation of Control Conditions The PST carries a subset of the region nodes in the PDG[9] [10]. For both representations holds that all child nodes are controlled by the same set of control conditions. Though the region nodes in the PDG are much more exhaustive than the information in the PST. In a CDG, as given in Figure 4.11e, all nodes with a single common control condition are already aggregated as child nodes of the same parent. The PDG also aggregates nodes that share more than one control condition by introduction of new region nodes that depend on these common control conditions.

The PST is obtained by packing nodes with exactly the same control dependencies. As it can be seen in Figures 4.11b, 4.11c, 4.11d and 4.11e, CDG and PST are similar in structure. Although the information contained in the PST is more suitable when determining the execution conditions than the node-based control dependencies as shown in Figure 4.11e. The reason therefore is that the control dependencies do not differentiate between different control-flow paths as long as the control flow is guaranteed to reach a certain node. SESE-regions are guaranteed to be entered by exactly a single edge. To construct SESE-regions from control dependencies it is therefore beneficial to consider edge-based control dependencies as presented in [31] instead.

[9] Program Dependence Graph

The PST creation also has an advantage in computation time compared to the control dependencies. The creation of the region nodes in the PDG is in $O(N^2)$ [10], whereas the PST may be built in $O(E)$ [19].

Node or Edge Bounded Regions The SESE-regions as provided by LLVM and the PST in [19] are represented as collections of nodes and subregions. In [19] these regions are bounded by a single incoming and outgoing edge on both ends. LLVM's notion of regions also requires them to be bounded by a single edge or, extending the regions definition in [19], it also identifies so called extended regions that may be turned into canonical regions by introduction of a new region-entry or -exit block and merging of the appropriate edges.

To illustrate the differences between the different types of SESE-regions the subsequent paragraphs examine in detail the example given in Figure 8a in [46] on page 108. The PST derived according to the definition in [46] for this example is shown in Figure 4.11b. The LLVM IR specified in Listing 4.2 resembles the second part of this CFG.

Figure 4.11a shows the CFG for this example. The regions identified by LLVM are separated by a colored background. Note that the region-exit block is not considered as a part of the region to prevent partial overlap between different regions. The edge labels in this graph correspond to those of Figure 8 in [46] for easier comparison.

LLVM's Complex Regions Listing 4.3 shows the output of LLVM's SESE-region detection when applied to the source from Listing 4.2. The 3 regions identified were { *entry=><Function Return>, v5=>t, v5=>v7* }. Note that the region *v5=>v7* is a complex region because *v7* is the target of the control-flow edges l and m as can be seen in Figure 4.11a. Complex regions as used by LLVM are control-flow structures that may be turned into canonical SESE-regions by aggregating several of their entry or exit edges into a new CFG node. As complex regions have not been part of the original definition for the PST in [19], these regions would have been merged with their ancestor region as shown in Figure 4.11d. Besides the exit blocks with more than one incoming edge, as in this example, the complex regions detected by LLVM also allow the sharing of the entry or exit block amongst several SESE-region like structures. In this case the complex regions need to be split up to gain the actual SESE-regions as shown in Section 4.3.3.

```
define void @regions(i32 %x) {
entry: br label %v5
v5: %b = icmp ne i32 %x, 0
    br i1 %b, label %v6, label %v7
v6: %b1 = icmp ne i32 %x, 0
    br i1 %b1, label %v7, label %v5
v7: %b4 = icmp ne i32 %x, 0
    br i1 %b4, label %v5, label %t
t:  ret void
}
```

Listing 4.2: Example source code to which region detection is applied

```
Region tree:
[0] entry => <Function Return>
{
  entry, v5 => t, t,
  [1] v5 => t
  {
    v5 => v7, v7,
    [2] v5 => v7
    {
      v5, v6,
    }
  }
}
End region tree
```

Listing 4.3: Regions as identified by LLVM

Properties of SESE-Regions Any SESE-region, be it node or edge bounded, has the following two properties (see [19], page 172 for edge bound regions) that simplify the predicate value determination:

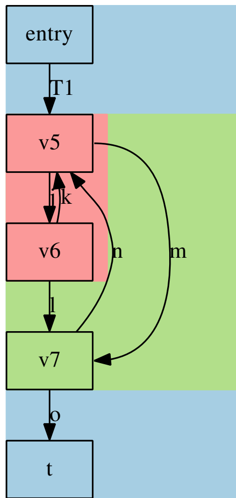
- The entry node dominates any node in the SESE-region including the exit node.
- Every node in the SESE-region is post dominated by the exit node.

Domination A graph node a is said to predominate [2], or dominate for short, another node b if all paths from the graph entry to node b must contain node a. Vice versa a node b is said to post-dominate node a when every path from a to any of the exit nodes must contain b. When, in addition, nodes a and b differ, $a \neq b$, the relation is called strict [9].

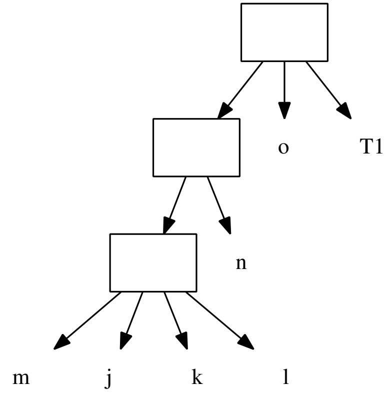
Execution Predicates Since the region entry dominates every node in the region, whenever the program execution reaches a node inside the region, the entry node will have been executed earlier. In terms of execution predicates that means, whenever the execution predicate for any node in the region holds when the execution reaches that node, then the execution predicate for the entry node must have held when the execution had passed this entry node before.

The reason therefore is that any predicate that belongs to a certain basic block is, when control reaches that particular basic block, supposed to evaluate to *true* iff the basic block is going to be executed. In conventional code this would always hold since control flow only reaches basic blocks that have to be executed. After application of the SP-transformation this is no longer the case. Since the region entry node will have been passed and executed whenever another node within the region is to be executed, the execution predicate for the region entry must have yielded true when it had been evaluated.

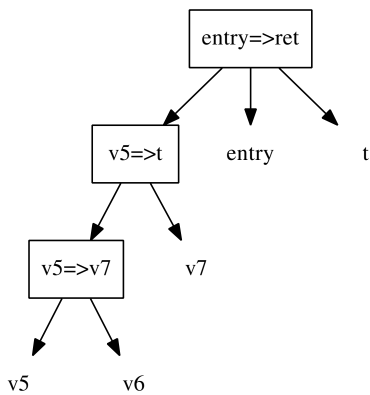
Also, since the entry node dominates all other nodes within the region, when the entry node's execution predicate evaluated to false, i.e., control in the original CFG



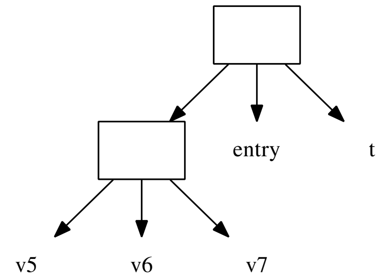
(a) Example CFG with derived regions



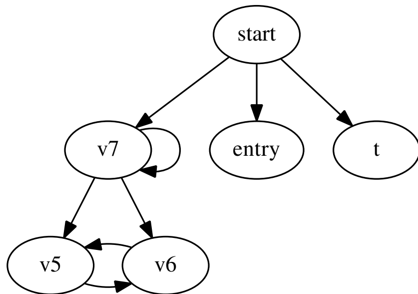
(b) The PST of Figure 4.11a as defined in [46]



(c) The PST for Figure 4.11a as calculated by LLVM



(d) The PST for Figure 4.11a as defined in [19]



(e) The CDG for Figure 4.11a

Figure 4.11: Comparing different types of PSTs

would not have reached the entry node the subsequent evaluation of all nodes within this SESE-region must also yield false.

Guarding For SESE-regions, as presented in the previous paragraphs, it is possible to create a guarding variable for every SESE-region. The execution condition of the entry block is therefore evaluated whenever the block is reached by the program control and the evaluation result is stored to the guarding variable. Since the preprocessing described in Section 4.3.3 guarantees that every basic block can at most be the header of one SESE-region, the entry block uniquely identifies a SESE-region. Later on this property is used to uniquely name the guarding variable for this region. E.g., a SESE-region with an entry block labeled r will have its guarding variable named σ_r. The guarding of SESE-regions presented here corresponds to the transformation rules for input dependent if statements from Table 1 in [36], where, in a similar scheme, a new guard is created for each side of if statements.

The execution conditions for all nodes that are contained within a SESE-region may be used in conjunction with σ_r, thereby allowing to remove these parts of the execution predicates that are subsumed by σ_r. Given that the execution predicates are constructed by propagating conditions along the edges of the CFG, it is guaranteed for any block contained within a SESE-regions that σ_r is the only condition they will ever have to consider from outside the region.

Aside from a simplification of the execution predicates this also allows an independent transformation in an arbitrary transformation order of SESE-regions. When transforming a SESE-region the transformation may make use of σ_r as part of predicate expressions before the outer SESE-regions have been transformed. The reason therefore is that it can be safely assumed that σ_r will be determined eventually in the course of the SP-transformation.

For the exit block of a SESE-region the same execution predicate that has been determined for the entry block of this region can be used. Since the entry block dominates the exit block and the exit block is post dominating the entry block, the exit block has to be executed iff the entry block has been executed before.

Expressing Predicates in the LLVM IR

The SP-transformation derives predicates from the CFG's structure that will ultimately be used in conjunction with the constant-time conditional operator [35] to guarantee a unique execution trace. The LLVM language specification [33] however does not support any types of predicates directly. Instead the predicates have to be expressed within the capabilities provided by the LLVM IR. In the following paragraphs two methods to express execution predicates by the means provided with LLVMs IR are described.

Metadata A possibility for storing the execution predicates along the LLVM IR is within the metadata section. Metadata are a part of the LLVM IR specification [33] and allow the storage of arbitrary information. Any IR instruction can have an arbitrary number of metadata references attached.

The results of the predicate calculation could be integrated with the LLVM instruction stream by applying a predicate metadata tag, that references the calculated predicate, to those instructions which are controlled by a predicate. Later in the compilation process these predicate metadata tags have to be evaluated and used to insert appropriate constant-time conditional operators into the instruction stream.

A downside of this approach is that the program will not execute correctly until the constant-time conditional operators have been created. This constitutes an obstacle to testing only partially transformed programs, which is something that has shown beneficial to verify the correctness of individual transformation steps.

Tagged Branches The usual way to prevent instructions from being executed in LLVM IR is by having branches that branch to some other basic block. Predicates may as well be expressed as branches that branch “around” a basic block consisting of predicated instructions when the predicate does not evaluate to *true*. These branches are subsequently called predicate branches.

The main advantage of using branches instead of metadata tags for representing the execution predicates is that they may be processed by the remaining compilation chain without any special treatment and the resulting code will still execute correctly. That, in turn, is useful to test the correctness of the individual transformation passes.

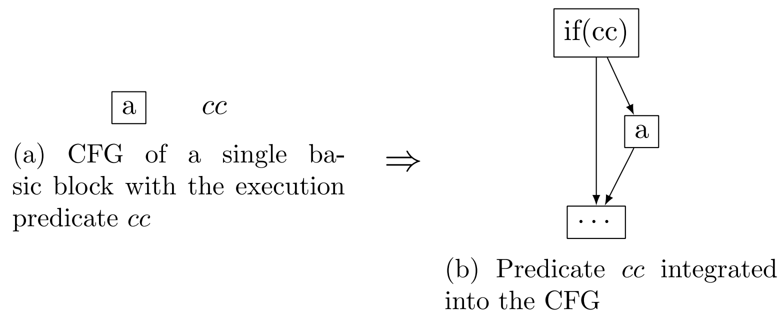
The new branches, that have been introduced to represent execution predicates, have to be replaced by the constant-time conditional expression later in the compilation process. They also require special treatment by some of the optimization passes to ensure that they do not get removed or otherwise modified. To enable the recognition of these predicate branches, a reference to a single named metadata node called *!sp_needs_conversion* is attached to their IR instruction. The optimization passes have been modified to ignore branches that reference this metadata tag. Additionally, branches that reference this metadata tag are, during code generation, replaced by conditional execution to mimic the behavior of the constant-time conditional operator.

When inserting a guarding branch into the control flow before control reaches the guarded basic block the execution predicate for this block is evaluated. When it evaluates to $false$ the execution control is not transferred to the guarded block. Any SSA values defined in the guarded block do no longer dominate any uses outside this basic block. Since this is a basic requirement for valid SSA-form [9] additional phi-instructions have to be added to the program. The control-flow merge, which immediately follows the guarded block, can be augmented by an additional phi-instruction and any invalid usages of the original SSA-value may then refer to this new phi-instruction instead. This ensures that the domination property required by SSA-form is satisfied again. The new phi-instruction selects between two incoming values. When control flow is incoming from the guarded block, the value defined inside the guarded block is selected. Otherwise, when the control flow skips the guarded block, the phi's value is set to an instance of $undef$. A detailed description on the handling of undefined values within LLVM can be found in [33].

An example of using a branch to represent the execution predicate is given in Figure

4.12b. This CFG shows that the condition *cc*, that has been attached as a predicate to basic block *a* as shown in Figure 4.12a, is now replaced by a branch.

A predicated block may have one or more outgoing edges. Additional measures not described here are required to preserve the program behavior in case of multiple outgoing edges. In case of a single outgoing edge, the control flow may be modified to join at the basic block that has previously been targeted by the single outgoing edge.



(a) CFG of a single basic block with the execution predicate *cc*

(b) Predicate *cc* integrated into the CFG

Figure 4.12: Integration of execution predicates into the CFG

4.3.5 Transformation

This section describes the implementation details of the transformation that is applied to input-dependent branches during the SP-transformation. The transformation is applied to input-data dependent branches at LLVM IR level. Subsequently this portion of the SP-transformation will be called branch transformation. The branch transformation assumes that the loop transformation as described in Section 4.2.4 and the general program preparations described in Section 4.3.3 have previously been applied.

Transformation Order The branch transformation processes SESE-regions individually, starting from the leaves of the PST. This order is beneficial for the testability of the transformation implementation, since it is not required to transform an entire program at once. Instead the program remains executable even when only partially transformed.

Loops When the transformation encounters a loop-header block that has previously been transformed by the loop transformation the entire loop is handled as an opaque block by this transformation, just like already transformed subregions are. Since the transformation is applied from the inner SESE-regions towards the outer anything contained within the loop will have been already transformed at this point.

Guard

SESE-regions may be transformed independently from each other whereby inner regions require to have an execution-predicate parameter provided by their outer region as described in Section 4.3.4. The current implementation passes this execution predicate in terms of an unsigned integer variable that is expected to be zero if the region should

be executed and nonzero otherwise. Later on this integer variable will be referred to as guarding variable.

The SP-transformation avoids the need to allocate a variable that stores the execution predicate for each SESE-region by combining the information of several SESE-regions execution predicates into a single guarding variable.

This paragraph describes how the guard value is manipulated at SESE-region bounds so that at the evaluation points the guard value reflects the execution predicates as described above. If the guard's value is nonzero before a SESE-region is entered it is incremented by one. If the guard's value is nonzero right after a SESE-region is left it is decremented by one. Assuming the availability of the constant-time conditional expression [35] this can be done in constant time as illustrated in Listing 4.4 and 4.5.

```
1 guard =
2     (guard!=0 # guard+1 : guard)
```

Listing 4.4: Incrementing guard in constant time

```
1 guard =
2     (guard!=0 # guard-1 : guard)
```

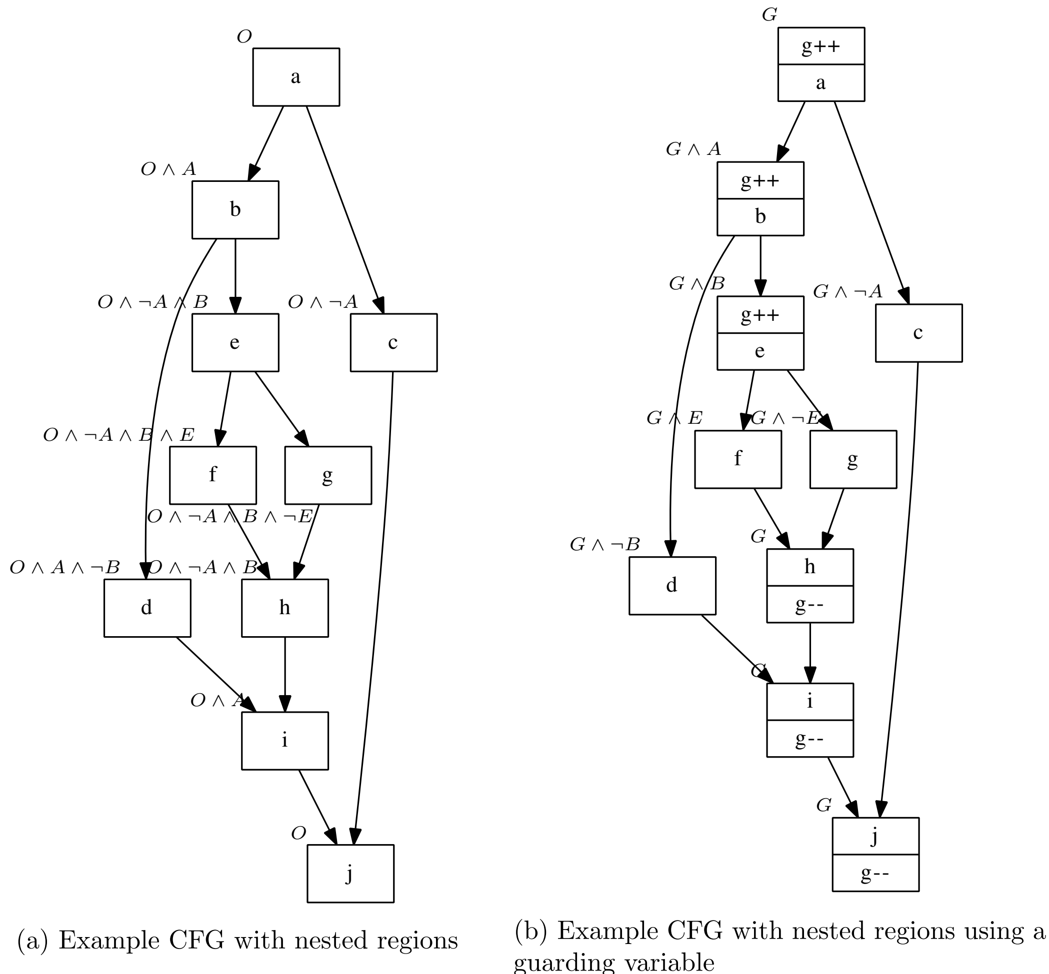
Listing 4.5: Decrementing guard in constant time

When treating the guard's value in this way the guarding variable has the same value after leaving a SESE-region as it had right before entering the SESE-region. When a region is disabled, i.e., the guard's value is larger than zero when the region is entered, the guard's value is incremented and decremented for every nested region encountered during execution. Essentially the guard's value is counting the nesting depth of SESE-regions where the execution predicate is false during execution.

Example Figure 4.13a is an example for a CFG containing nested regions. The nodes in the CFG are attributed with their execution predicates. Nodes in this example are named lowercase. When a node has multiple outgoing edges, the condition that controls which of these edges is executed is represented as an uppercase letter corresponding to the nodes name. E.g., node a has two outgoing edges where one is executed if A holds and the other if $\neg A$ does. In a CFG it is common to label the former edge T and the later F, but the graphs seem already a bit cluttered so these labels are omitted. Assuming the entire graph is embedded in a larger graph, O names the predicate induced by the outer graph.

Figure 4.13b is a graph with the same structure, but now the execution predicates make use of the guarding variable. The predicate term G represents a check of the guarding variable g for equality to zero. The evaluation of predicate G is expected to yield $true$ if the block it is attached to should be executed. The outer regions, if existing, are expected to use the guarding variable in the same way. At the region entry G is expected to already incorporate any predicates from outer SESE-regions that control the execution of the nested SESE-region.

The entry and exit blocks to SESE-regions are now preceded and succeeded by guard increment and decrement instructions. These are assumed to be implemented in a constant-time fashion. The application of the execution predicates has to be refined for blocks manipulating the guard's value. Blocks preceded by a guard incrementing



(a) Example CFG with nested regions

(b) Example CFG with nested regions using a guarding variable

Figure 4.13: Using a guard variable with nested regions

instruction execute this increment instruction iff the execution predicate determined for this block evaluates to $false$ when the block is entered. The remainder of this block has G as the only predicate to consider. Likewise the decrement instruction has to be executed only if the predicate G of this block evaluates to $false$. In summary, the guard increments and decrements are executed only for code parts that should not be executed.

Nesting Depth The current implementation makes use of a 32-bit wide integer allowing $2^{32}-1$ levels of region nesting when every region increments the guard's value by 1. Given the cycle equivalence [19] property of SESE-regions the number of increments and decrements within a loop iteration are always the same, so loops cannot use up the guarding variables value space regardless of the number of loop iterations executed.

4.3.6 Reordering the Control Flow

Once, according to Section 4.3.4, the execution predicates have been established for all blocks within a SESE-region the control flow within the SESE-region is reordered so that a unique execution trace is guaranteed. This is done by chaining the basic blocks in topological order.

Topological Order The CFG is restructured so that the resulting unique execution trace visits the nodes in topological order [17]. Since nested SESE-regions and loops are considered as opaque nodes during the transformation of a region, the graph considered in a single transformation step is actually a DAG. For any DAG the determination of the reverse postorder yields a topological order.

SSA-form requires that any variable use is dominated by its definition, as described in [9] on page 454f. Fortunately rearranging the control-flow nodes in a topological order keeps this property.

The reordering is done by removing branches from each node inside the currently transformed region and replacing it by a single unconditional branch to the following node, or already converted region, in topological order. As an optimization, some of the input-independent branches may be preserved, as is described later on. This can reduce the program's execution time.

Reordering – For each block *b* in topological order:

1. If necessary, rewrite phi-instructions in *b*.
2. Replace branches outgoing from *b*.
3. Insert code that evaluates the execution predicate of block *b*.
4. Create a predicate branch to guard the execution of *b*.

Execution Predicates In the current transformation implementation, the execution predicates are determined along with the branch replacement procedure. Whenever a basic block, whose execution is controlled by a predicate, is transformed, the SP-transformation inserts an instruction stream before the basic block to evaluate its execution predicate. Since the blocks are considered in topological order by the implementation, it is guaranteed that the execution-predicate information for the current block is already completely available. The evaluation result always has the type *i1*, a one bit wide integer, which is the common representation of boolean values in LLVM's IR. The evaluated value is expected to equal to *1* when the basic block should execute or equal to *0* if it should not. This value is passed to any successor blocks that depend of the execution predicate of the current block.

Whenever a branch is encountered by the transformation, the execution predicate of the branch's target blocks is extended to reflect a transfer of control along this edge in the original program. For unconditional branches the predicate of the originating block is added to the predicates of the target block by a logical *OR* operation.

When the branch is conditional the originating block's predicate is combined by a logical *AND* operation with the branch predicate, the resulting predicate is added to the target block's predicate by a logical *OR* operation.

Optimization In its simplest form the SP-transformation would replace all branches, be they conditional or not, in the program by a branch to the next basic block in topological order, resulting in a chain of basic blocks and transformed subregions which are executed in sequence. The optimization described here preserves some conditional branches in the transformed program. Where possible without losing the property of a unique execution trace the transformation omits branches from the replacement procedure, resulting in faster execution times of the transformed program.

These branches must have previously been determined to be input-data independent. Additionally, none of the branches preceding the branch to preserve in topological order must have been replaced within the currently transformed SESE-region. This condition guarantees that the preserved branch does not skip any blocks that are expected to be executed.

If a preceding branch had been replaced by a branch to its successor in topological order, the control flow passes basic blocks that would not be executed in the original program. In the SP-program these blocks do not execute even if execution control is passed to them because their guarding condition evaluates to $false$. When such a block is terminated by a conditional branch, this branch also would not execute in the original program. Keeping this branch in the transformed program, without further measures, could skip blocks that should actually be executed.

In the transformation described here a branch is only allowed to be kept when the execution condition of the branching block being $false$ implies that the execution condition for all control dependent blocks is also false. The simple condition that no previous branch in the SESE-region may have been transformed is sufficient since CFG parts that fulfill this condition form their own SESE-subregion.

Creating Guarding Branches

In the following transformation step guarding branches are created. These skip individual basic blocks when their execution predicate does not hold. These conditional branches, along with their condition calculations are used to represent execution predicates because the LLVM IR does not provide means to directly represent predicates. Within Section 4.3.4 one can find a more elaborate description on the use of branches to represent execution predicates. The introduction of these branches creates a control-flow join immediately after each predicated block. These joins are used to rewrite the phi-instructions that have been left in an invalid state by the transformation steps so far, as is shown in detail in the following section.

Any SSA variables defined in the guarded blocks are rewritten to be only defined when the block is executed. This process includes introduction of new phi-instructions in the join block immediately following the guarded block and phi-instructions that keep the value during additional loop iterations introduced by the *SP loop transformation* pass.

Rewriting Phi-Instructions

The replacement of control-flow branches by an unconditional transfer of control to the next basic block in topological order reduces the number of branches targeting the former successor blocks. When such a basic block contains phi-instructions, these instructions are no longer valid since at least one of the incoming edges they refer to do no longer exist. The following paragraphs present a strategy to rewrite these phi-instructions to obtain a valid SSA-program again.

LLVM's phi-instructions correspond to ϕ-functions of the SSA form as described in [9] on page 457ff. Both expect a list of alternatives from which a value is chosen depending on the branch origin through which control flow reaches the node/block containing the ϕ-function/phi-instruction. A notable difference between the two is that the ϕ-function expects a common order amongst predecessor blocks and function operands, whereby the phi-instruction expects value pairs that identify the block from which control may be transferred and the value to select in this case. From here on the description of the transformation will focus on LLVM's phi-instructions since that is what the implementation is using.

The rewriting of LLVM's phi-instructions is done along with the reordering of the control flow in topological order of the CFG. Thus all basic blocks referenced in a phi-instruction are guaranteed to have already been processed during program execution. Loop header blocks do not need to be considered by this transformation step since every loop has a pre-header inserted during program pre-processing. The pre-header carries any phi-instructions that select values when an edge from outside the loop to the loop-header is executed.

Edge Condition During rewriting of the phi-instructions a phi-instruction is split up into two or more phi-instructions which pick values according to the guarding expression of the predecessor blocks. In order to generate these phi-instructions empty basic blocks are created that are guarded by the condition required for the values to be selected by the phi-instruction. This condition is the edge condition of the incoming edge referred in the original phi-instruction. It is the guarding condition of the previous incoming block conjuncted with the condition required to execute the edge targeting the node containing the original phi-instruction. For blocks that branched unconditionally to the phi-instruction's block the branching condition is assumed as always true. The same is applicable for input independent branches which are not removed by this transformation.

Setphi Block In order to update the value defined by phi-instructions that require rewriting, empty basic blocks have been inserted. These blocks are subsequently called setphi blocks. One of these blocks has been created for each incoming edge of the original phi-instruction. These blocks have to be integrated into the control flow so that each of them is dominated by the block that originated the incoming edge before guarding. This condition ensures that the definition of the value that had been picked by the original phi-instruction is dominating the setphi block. After the setphi blocks have been created and placed appropriately they are guarded by their corresponding edge condition.

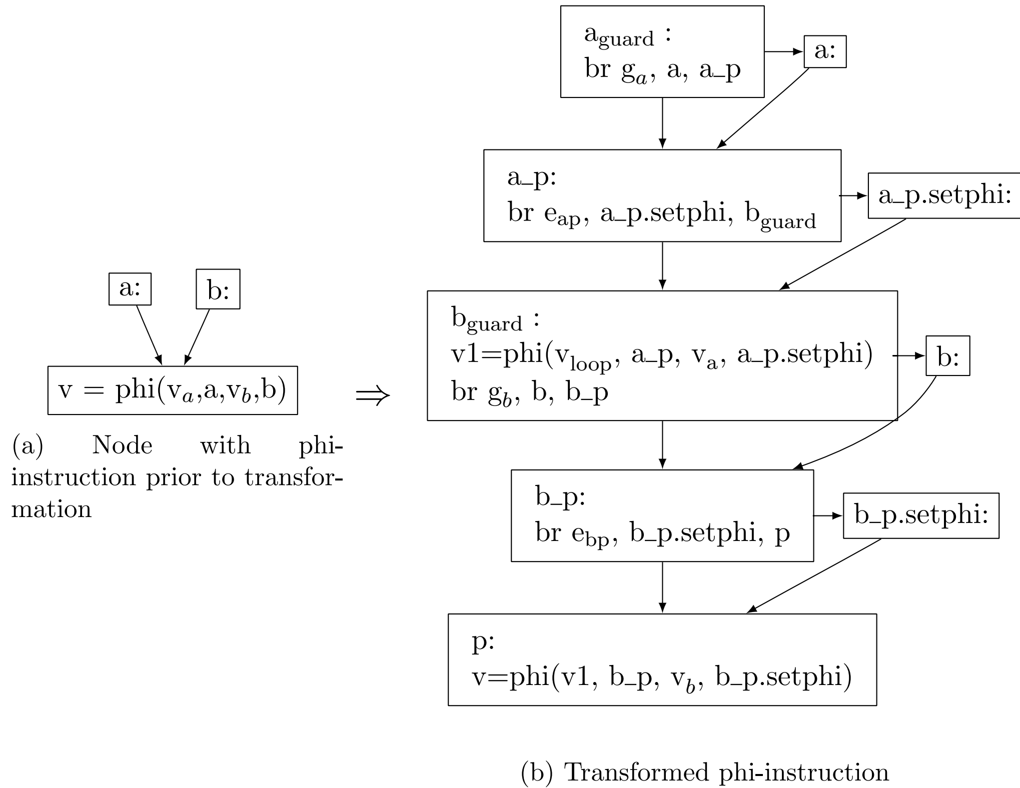
In the current implementation any setphi block is always placed immediately after the block that originates the incoming edge to the phi-instruction in the original program. In this way the evaluation result for the common part of the guarding expression of both blocks is more likely to be kept in a processor register during code generation which is saving a round trip to stack memory.

Example The CFG in the example given in Figure 4.14a contains a phi-instruction that needs to be rewritten after the blocks originating the incoming edges have been transformed. This phi-instruction defines the variable v by selecting v_a when the control flow is transferred from block a or v_b when the control flow is transferred from block b.

The transformation result is depicted in Figure 4.14b. Two new guarding blocks, a_{guard} and b_{guard} have been introduced. These blocks contain branches that control the execution of the blocks a and b depending on the result of the predicate evaluation, whose result is held in g_a respectively g_b. The evaluation result of these edge conditions is expected in e_{ba} and e_{bp}. Note that the instruction stream required to evaluate all these conditions is omitted from this example. Also any additional phi-instructions that may be required by the introduction of predicates by branches, as described in Section 4.3.4, are not shown here.

The new basic blocks *a_p.setphi* and *b_p.setphi* contain no instructions. They are required by the following phi-instructions to distinguish between different incoming paths. Later in the code generation process the phi-instructions will be replaced by register assignments that will partially be placed into these currently empty basic blocks.

The determination of the value of v is split up into two phi-instructions. For phi-instructions with more than two incoming values the same scheme can be applied by creating an additional phi-instruction for every additional incoming edge. Every phi-instruction selects between the value produced by its predecessing phi-instruction or one of the values the original phi-instruction selected from, depending on the edge condition that has been determined for the incoming edge of the original phi-instruction. The initial phi-instruction in this chain of phi nodes cannot use the value produced by its predecessor since there is none. Instead it is initialized with $undef$ or, if the value produced is contained within a SP-transformed loop, a reference to another phi-instruction that preserves the produced value amongst any additional loop iterations introduced by the transformation. An example therefore is v_{loop} in Figure 4.14b.



(a) Node with phi-instruction prior to transformation

(b) Transformed phi-instruction

Figure 4.14: Example of a phi-instruction that is rewritten during the transformation

4.4 Function Calls

This section describes how function calls are handled by the current implementation of the SP-transformation.

In [36], Table 1, transformation rules for procedure definitions and procedure calls are given. The following paragraphs describe these transformation rules and explain how the current implementation realizes them.

With the transformation rules in [36], Table 1, for any procedure definition p an alternate single-path transformed procedure named *p-sip* is created. The single-path version of the procedure has its parameter list extended by an additional parameter carrying the procedure's execution condition, called *pcnd*. Procedure calls for which the precondition σ is known to evaluate to true are left unmodified. All other procedure calls are replaced with a call to their single-path variant, with σ passed as the first argument.

Implementation differences Contrary to the specification in [36] the current implementation transforms every procedure definition, with some exceptions specified later in this section, into its single-path variant. Procedure calls where the guarding condition is known to be *true* still have to call the SP-transformed procedure. In that case the current implementation is passing *0* as the guarding argument. Note that the current

implementation considers a value of *0* to represent a fulfilled execution predicate. These cases obviously may increase execution time.

4.4.1 Passing the Guard Value at Function Calls

With any function call to one of the SP-transformed procedures the execution predicate has to be passed as a function argument. The implementation is currently adding a guard argument in front of the parameter list, exactly as suggested in [36]. The following paragraphs give a short overview about some implementation alternatives for passing the guard value, that have been considered during implementation and their implications.

Global Variable Functions may expect the current guarding value in the global variable space. For multi-threaded environments a thread-local storage location shall be used to guarantee that the individual threads do not interfere.

Advantages Passing the guard value in the global variable space has the advantage that the function signatures remain unmodified. When the global guard variable is additionally initialized correctly, e.g., by placing it in a zero initialized memory section when a guarding value of zero represents a fulfilled execution predicate, the SP-transformed functions may be used by clients that are completely unaware of the SP-transformation.

Disadvantages At function calls, accesses to the global memory are required even when a free register would be available. Before any function call the global variable has to be updated to reflect the current state of the execution predicate. The called function likely has to load the global guarding value at least once.

Function Argument The guard value may be passed to functions by means of an additional function argument. The callee is not required to restore the value before it returns since the value is duplicated at the function call. This approach requires a modification of the function signature.

Advantages Function argument passing is a common operation and hence ABI specifications usually try to keep the runtime overhead small. E.g., in the AAPCS[10][32], which is part of recent ARM ABI definitions and which is used in the current SP-transformation implementation, the registers r0-r3 are used to pass arguments.

Disadvantages The function signature is modified. All clients of a SP-transformed function have to be SP-aware and pass the additional guard argument with any function call.

Register A register could be used to hold the guard value across function calls. This approach is most reasonable when the guarding value is stored in a reserved register that is not used otherwise, since that may be simple to implement.

Advantages No additional operations are required at function calls, when the guarding value is located in a reserved register.

[10]Procedure Call Standard for the ARM Architecture

Disadvantages Registers are scarce.

4.4.2 Implementation Details

This section describes the transformation of function definitions and function calls in the current implementation of the SP-transformation. Everything described in this section refers to a program representation in LLVM IR since this is the representation the current implementation is working with.

Function Declaration Any function declaration, with some exceptions described in paragraph *Ignoring Functions*, encountered during compilation has an additional argument added. This argument is called *guard* and is added before the first argument to the existing argument list. The type of the new argument is a 32-bit wide integer, *i32* in LLVM's notation. Prepending the guard argument to the argument list saves from having to implement special handling for functions with a variable number of arguments.

Function Definition Function definitions have the function signature expanded by an additional guard argument in the same way as declarations. Whenever the branch and loop transformation refers to the guard value at the outermost SESE-region this additional argument is what is actually referenced.

Function Call Every function call in the program, with some exceptions listed in the following paragraph *Ignoring Functions*, has the current value of the guarding variable added as the first parameter. The loop and branch transformation have to ensure that the guard's value is updated to reflect the current execution predicate before executing any function call.

Calls to Compiler-RT

Compiler-RT[11] is a set of compiler-support routines developed along the LLVM compiler. These routines implement some of the more complex algorithms required to lower the LLVM IR to machine instructions, e.g., floating-point arithmetic for target architectures that do not contain an FPU[12]. This library is LLVM's equivalent to the GCC[13] low-level runtime library *libgcc* and implements mostly the same methods.

Calls to Compiler-RT require some special handling by the SP-transformation since they are not in place in the IR representation on which most of the SP-transformation process works.

Function Definition The conversion of function definitions of Compiler-RT functions is done in the same way as all other functions. A guard-value argument is prepended to the argument list and the input-dependent branches are transformed. Since the method

[11] Real-Time / Runtime
[12] Floating-Point Unit
[13] The GNU Compiler Collection

calls to Compiler-RT functions have not been in place when the dataflow analysis has been executed all arguments to functions contained in Compiler-RT are conservatively considered input-data dependent.

Function Call Any calls to Compiler-RT functions are required to pass the additional guard argument. Therefore the instruction lowering phase, that introduces these calls, has been modified to add an additional *0* as the first argument. As a result the function calls are valid again but the guard argument has no effect yet. For function calls that should be executed unconditionally, i.e., where the execution predicate is always *true*, there is nothing more to do.

Calls to functions in Compiler-RT, that should only execute under a certain execution predicate, are created during the lowering of IR instructions. But these calls do not consider the execution predicate in any way. The requirement for a certain execution-predicate condition is expressed by a guarding branch marked by the metadata tag *!sp_needs_conversion* as created by the branch transformation. These branches are removed later on during the *machine-instruction predication* pass. Additionally, this pass also recognizes the Compiler-RT calls that need an additional guarding argument and replaces the *0* passed as the first argument by an evaluation of the execution condition. When the execution predicate evaluates to *true* a value of *0* is passed, when the execution predicate evaluates to $false$ a value of *1* is passed.

As an example the machine code of method *frexp()* is provided in Listing A.1. It shows a call to *__nedf2* without any SP-modifications to the function call. Listing A.2 shows the same function call with a value of *0* passed as the guarding argument in register *r0* as one can see in line 7. This function call would already be valid in a SP-transformed environment when the execution predicate is known to be always *true*. For all other method calls this is only an intermediate step. Later in the compilation phase, during the replacement of the predicate branches by actual predicates, the value passed as the guard argument is updated to reflect the current predicate. For the example this is shown in Listing A.3 in the lines 10 and 15.

Ignoring Functions

Some functions do not have an additional guard parameter added during SP-conversion. These are the functions named *_start* and *main* which are expected to carry initialization code.

Additionally, any function that has the function attribute *__sp_ignore* attached is also excluded from the addition of the guard parameter. This is true for the function declaration, which is not touched by the SP-transformation at all, as well as for any calls to this function. When a function definition has the attribute *__sp_ignore* attached, for correct behavior it is required that any declaration of this function throughout the entire program also has *__sp_ignore* attached. An example for a function declaration making use of this function attribute is given in Listing 4.6.

```
1 void debug_print() __attribute__((__sp_ignore));
```

Listing 4.6: Function declaration using the __sp_ignore attribute.

4.5 Code Generation

This section describes the final phase of the SP-transformation. It takes place during the compiler's code generation. The current implementation targets the Thumb-2 ISA[14] on an ARM Cortex-M3 based micro controller. Any machine-code related parts that follow in this section refer to this this type of micro controller.

At first Section 4.5.1 gives an overview how the code-generation phase is implemented in LLVM's backend. The following Section 4.5.2 outlines why the SP-transformation replaces branches in the program with conditionally executed machine instructions. The implementation of this replacement is described in the Section 4.5.3. This implementation is split into two compilation phases which are separately executed before and after the register allocator. These phases are explained in detail in Section 4.5.4 and Section 4.5.6. The pseudo instructions which are inserted in the command stream as an immediate step are separately described in Section 4.5.5.

4.5.1 Code Generation Overview

Code generation is the process of transforming the program from the LLVM IR into the target representation. In the current implementation of the SP-transformation this conversion targets Thumb-2 machine instructions. The following list gives an overview of the individual transformation steps executed during code generation.

This list contains passes included in any stock LLVM version, as well as passes that are unique to the LLVM variant that generates SP-code. Some of the transformation steps inherited from LLVM require slight modifications which are described in the following sections.

Code generation steps:

1. IR preparation
2. Insert Exception Handling code
3. Instruction Selection
    - IR to Selection-DAG conversion
    - Instruction Selection
    - Instruction Scheduling
4. SP-Predicate pre-RA
5. Phi-Elimination
6. Register-Allocation
7. SP-Predicate post-RA
8. Post-RA Scheduler
9. Block-Placement
10. Print Assembler-Instructions

[14] Instruction Set Architecture

IR Preparation

The IR-preparation phase subsumes several basic optimization that are applied to the IR-code to simplify the further transformation steps. E.g., removal of basic blocks that could never be reached, so called dead blocks. The following paragraphs describe modifications to these transformation steps that have been conducted during the implementation of the SP-transformation.

Block Merging During the IR-preparation blocks that are connected by unconditional branches are merged. The later SP-code generation steps create block structures that are expected to be retained during the transformation. So, after the initial block-merging phase, no further basic blocks are merged during the SP-transformation. Therefor some block merging passes in LLVM had to be partially disabled. These passes have been extended so that they do not merge blocks that have the metadata tag *!dont_merge* attached to any instruction inside the blocks. The earlier SP-transformation steps apply this tag to any blocks they want to keep separated.

Tail Duplication The tail duplication passes, *early-taildup* and *tail-duplicate*, had to be disabled in order not to interfere with SP code generation. Tail duplication would move phi-instructions down the CFG, essentially reverting the region simplification done in preparation for the SP-transformation. This region simplification is described in Section 4.3.3.

Optimize Compare Expression The *OptimizeCmpExpression* pass is part of the IR preparations that are applied prior to code generation.

This optimization is called for every compare instruction. It checks, if the result of the compare operation is used by an instruction outside of the basic block in which the compare resides. If so, a copy of the compare instruction is created at the beginning of the basic block in which the depending instruction is located. After duplicating the compare instruction for every one of its dependent instructions the original compare instruction is no longer used and may be removed from the program.

The execution predicates introduced by the SP-transformation are expressed as conditional branches based on the resulting values of such compare instructions. This is necessary because LLVM's IR does not directly provide means to attach execution predicates to program parts. When transforming the conditional branches into predicates the SP-transformation expects a continuous sequence of blocks which has been carefully prepared by the earlier transformation steps. Letting the *OptimizeCmpExpression* pass duplicate the compare instructions could possibly modify the processor-state register which would later on, during the register allocation, require additional processing to save the state register's current value. The reason therefor is that the SP-transformed program makes heavy use of the state register to hold the current execution predicate. As a consequence this transformation would increase the execution time in the SP-transformed program.

The earlier SP-transformation passes attach the metadata attribute *!dont_move* to any compare instructions that are not expected to be relocated. This attribute is then evaluated by the IR preparation.

IR to Selection-DAG Conversion

The so-called Selection-DAG is a dataflow oriented program representation on which instruction selection is performed. This is done by matching portions of the Selection-DAG to the dataflow DAG representations of the available machine instructions. The following paragraphs show where the transformation of the IR into the Selection-DAG has been modified to consider the requirements of the SP-transformation.

Emit Branches for Logically Combined Conditions During the Selection-DAG construction the conditions of all branches in the IR are analyzed. When the branch condition is logically combined by an and- or or-operator this branch is split up into several branches. This is in particular true for the conditions used by the guarding branches which are a carefully crafted sequence of conjunctions and disjunctions as described in Section 4.3.4.

Splitting a basic block that is terminated by a branch that depends on a combined condition into two or more basic blocks requires the introduction of new branching instructions. These new branches may be input-data dependent. But since the identification of input-dependent branches is executed in a much earlier stage of the SP-transformation process, these newly created branches would be ignored by the remainder of the SP-transformation and end up in the generated machine code. The resulting code in turn would not fulfill the requirements of single path code due to these branches.

An example of a typical IR fragment for code that is generated during execution-predicate calculation is given in Listing 4.7. Listing 4.8 shows the machine code generated out of the code from Listing 4.7 by a stock version of LLVM. Note the additional input-dependent branch in line 4. Finally Listing 4.9 shows the sequence of machine instructions generated by a LLVM variant that is modified to generated SP-code. From the absence of branches in the code follows that all the logical operations are executed even if the result of the operation is already implied by one of the preceding operations.

```
1  if.else.branch:
2    %10 = load i32* @__guard1
3    %11 = icmp ugt i32 %10, 0
4    %12 = xor i1 %cmp11join_phi, true
5    %not_guard_and_not_X = and i1 %11, %12
6    br i1 %not_guard_and_not_X, label %if.else15, label %if.else.inc_guard
```

Listing 4.7: IR of a branch that is expected to end up in a single basic block

```
1  .LBB2_10:
2          ldr     r12, [r2]
3          cmp     r12, #0
4          beq     .LBB2_12
5  @ BB#11:
6          eor     r3, r3, #1
7          cmp     r3, #0
8          bne     .LBB2_13
```

Listing 4.8: ARM code that is generated by the stock LLVM compiler

```
1  .LBB2_10:
2          ldr     r12, [r2]
3          cmp     r12, #0
4          eor     r3, r3, #1
5          mov     r12, #0
6          movne   r12, #1
7          tst     r12, r3
8          bne     .LBB2_12
```

Listing 4.9: ARM code with logically combined conditions preserved

Instruction Selection

During instruction selection some DAG patterns are replaced by calls to functions provided by the Compiler-RT library. These are mostly patterns requiring more complex implementations on the particular target. During the creation of a SP-transformed program all functions in the Compiler-RT have been modified to expect a guard parameter. The creation of function calls to functions contained within the Compiler-RT has been modified to provide this additional guard parameter. In the current implementation a dummy parameter of constant zero is added. The actual guard value is provided during instruction predication later on.

4.5.2 Turning Branches into Predicates

The key modification implemented in the code generation phase, which enables the generation of SP-code, is the replacement of branches by predicated instructions. The idea behind that is to approximate the behavior of the constant-time conditional expression [35] by these predications.

Hardware Predication Support

The implemented approach to replace branches by predicating all instructions dependent on these branches requires appropriate hardware support, i.e., all instructions must support conditional execution.

Cortex-M3 Predication Support Since the current implementation targets a Cortex-M3 based microcontroller, a closer examination of the instruction timing and predication support, conditional execution in ARM terms, on these types of micro-controllers follows here. Note that the terms predicated execution and conditional execution are used interchangeable here.

The Cortex-M3 is an implementation of the ARMv7-M [4] architecture profile. The execution timing for the Cortex-M3 architecture is described in [8], Section 3.3. Conditional execution is described in [7], Section 3.3.7.

The APSR[15] contains 4 state flags which are updated by certain arithmetic oper-

[15] Application Program Status Register

ations. These are Negative, Zero, Carry and Overflow, abbreviated by N, Z, C, and V. Note that the register model of the current ARM back-end implementation within LLVM still uses the label CPSR[16] for this register. Instructions may be executed, based on the current value of these four condition flags. In the Cortex-M3 ISA, which is limited to Thumb instructions, this is realized by prefixing the instructions that should execute only under a certain condition by an IT instruction as shown in [7], Section 3.9.3. A set of 14 execution conditions is defined in [4] Section A6.3. These conditions are directly derived from one or more of the 4 state flags described above. In assembler code, when using UAL[17] instruction syntax, each of these 14 conditions is represented by a two letter suffix which is also listed in this table.

Execution Timing Although the Cortex-M3's architecture is rather simple compared to the larger processors available nowadays, the determination of the execution timing is not that simple anymore. The execution time of certain instructions depends on the systems current state. Ideally, to mimic the constant-time conditional expression by the use of predicated instructions, the execution of each instruction would take exactly the same time regardless if the instruction's predicate is fulfilled or not.

For Cortex-M3 based microcontrollers this is clearly not the case as one can tell by examining the instruction set summary table from [8], Section 3.3.1. In general instructions are specified to execute in between 1 and 12 clock cycles, some varying dependent on conditions like the type of the preceding instruction, pipeline stalls or the data processed. Under certain conditions instructions may be folded onto their preceding instruction and require no execution time at all. Instructions waiting for an interrupt or event may take even longer than 12 cycles. When instructions are predicated and their condition evaluated to $false$, they usually take one clock cycle to execute. As a result, the execution time of blocks containing predicated instructions is expected to vary, depending on the condition state.

4.5.3 Implementation

In this section the preparations that are required to enable the replacement of branches by predicated instructions are described. These preparations are mainly required to reserve registers so that they are not assigned by the register allocator. These preparations are performed before register allocation in the pass called *SP-Predicate pre-RA*.

Preconditions After the preprocessing steps done by the loop and branch transformations it is guaranteed that all input dependent branches are branching around exactly one basic block. This prevents any nesting of input dependent branches as shown in the example given in Figure 4.12b. Additionally the input-data dependent branches are guaranteed to have the *!sp_needs_conversion* metadata tag attached. These transformations have been done on the IR of the program, so these guarantees are primarily

[16] Current Program Status Register, deprecated synonym for APSR
[17] ARM Unified Assembler Language

asserted on this level of representation. Since the instruction selection, which has been executed before, works on a basic block level the program structure remains unmodified, keeping these properties also in the machine-instruction representation.

Existing Predicates The subsequently described transformation is required to assign a predicate to any instruction inside guarded blocks. In the targeted architecture an instruction may have at most one execution condition assigned. When the instructions in the input program already make use of conditional execution the transformation has to derive a condition that combines the previously used condition with the SP execution condition. To avoid this construction of combined conditions the transformation prefers input programs that do not make use of conditional instructions. The transformation is implemented to be correct in the case of prior existing predicates but the additionally required predicate handling will result in longer execution times.

If-Converter Pass LLVM's IR does not directly provide conditionally executed instructions. When a program is compiled by the stock LLVM compiler, any use of predicated instructions in the resulting program is introduced by passes working on the machine-code representation. The pass that introduces most of the predicated instructions when code is generated by the stock LLVM compiler for an ARM target is the *If Converter* pass. This pass is originally applied after instruction selection.

In the modified LLVM compiler used for SP-transformation the *If Converter* pass is disabled. With this pass disabled, the generated code is mostly predicate free. Note that this pass implements a transformation that is similar to the transformation described here but with a focus on performance improvement and other varying details, e.g., it ignores basic blocks that already contain predicates.

4.5.4 Transformation before Register Allocation

The *SP-Predicate pre-RA* pass inserts pseudo instructions into the command stream with the purpose to reserve one or more registers for later use by the *SP-Predicate post-RA*[18] pass. It will use these registers to store conditions and the state of the condition flags in the *APSR*. The pseudo instructions inserted are described in Section 4.5.5.

Transformation Realm The *SP-Predicate pre-RA* transformation works on a subset of the CFG that consists of three or four basic blocks. A *branch* block, the two targets of this branch, which are called *true* and *false* block here. And a *join* block which is the common, unconditional, branch target of the *true* and *false* block. When the *branch* block executes, the *true* block is executed next if the branching condition is fulfilled. Otherwise the *false* block is executed next. The transformation also allows structures that omit either the *true* or *false* block, thereby the *join* block is a direct target of the *branch* block.

[18] Register-Allocation

When neither the *true* nor the *false* block contain any instructions that modify the APSR nothing has to be done for these blocks.

Reserving Registers In the following the insertion of pseudo instructions into the command stream is described. These pseudo instructions add additional register references for which later on the register allocator will assign physical registers.

In case there are any instructions modifying the *APSR*, an *SP_SAVE_PRED* pseudo instruction is inserted immediately before the branching instructions in the *branch* block.

In both, the *true* and the *false* block, after each instruction that may modify the *APSR* an *SP_SAVE_PRED_COMBINED* pseudo instruction is inserted when the just defined predicate is referenced by a later instruction.

The saved predicates are required whenever an already predicated instruction is encountered in the instruction stream. An *SP_EVAL_PRED* pseudo instruction is added to the program immediately before these predicated instructions. These new instructions are responsible for restoring the APSR to the state it would have had without the entire transformation.

In the *join* block a phi-instruction is inserted which references the predicate registers value regardless if the control flow enters from the *true* or the *false* block. This phi-instruction ensures that the predicate registers are referenced and are not declared dead and removed during register allocation.

4.5.5 Pseudo Instructions

The following paragraphs describe the semantics of the pseudo instructions inserted by this transformation. Since registers are still expected to be in SSA-form when these pseudo instructions are created, the same register can not be assigned twice. Instead the pseudo functions take a reference to the register containing the previous value, subsequently called *predreg*, and write the result into a newly allocated virtual register. The register with the previous value is never used after this point which enables the register allocator to reuse the same physical register.

<dst> = SP_SAVE_PRED <pred> Evaluates if the current state of the APSR matches the predicate supplied by *pred*. The evaluation result is stored to the register *dst*. This instruction is commonly used to retain the branch condition of a branch instruction that is removed during this transformation.

<dst> = SP_SAVE_PRED_COMBINED <predreg> <pred> Evaluates if the current APSR matches *pred* and combines the result with a previous saved predicate. The register *predreg* is assumed to contain a predicate state that has been saved by an earlier *SP_SAVE_PRED*. The result is stored to register *dst*. *dst* is expected to contain both, the combined and the previous uncombined evaluation result and may be used appropriately by subsequent *SP_EVAL_PRED* instructions.

SP_EVAL_PRED <predreg> <restore_carry> <pred> Restores the value of the APSR based on the information in *predreg*. The register *predreg* is assumed to carry the result of a previous *SP_SAVE_PRED* or *SP_SAVE_PRED_COMBINED* instruction. When the boolean argument *restore_carry* is set, the carry flag of the APSR is restored based on the information in the register *predreg*. This may be necessary because all APSR modifying operations inserted here do also clobber the carry bit.

4.5.6 Transformation after Register Allocation

During register allocation the pseudo instructions that have been inserted before, as described in Section 4.5.4, had physical registers allocated. In the post-RA transformation described here these pseudo instructions are now replaced by actual machine code. This step is, of course, highly target-architecture specific. A short description of the implementation targeting a Cortex-M3 based microcontroller follows. In addition to replacing the pseudo instructions this pass removes the input dependent branches from the program, appends the branch target blocks to the branching block and predicates the instructions in these blocks as necessary.

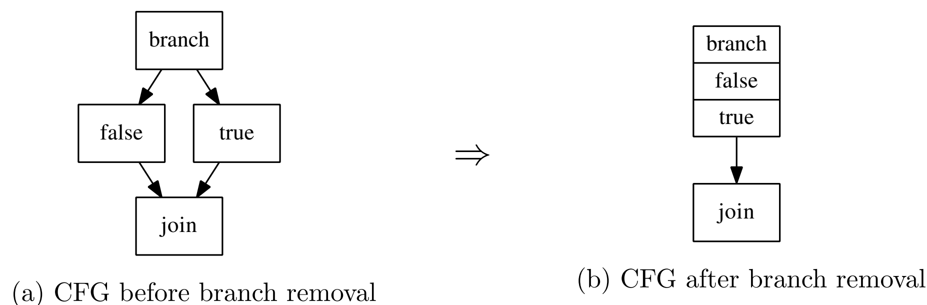
This transformation is again working on a *branch* block, its two target blocks which are called *true* block and $false$ block here and their common target called *join* block.

Removing Branches

As the first step the branching code is removed from the *branch* block. The branching condition is preserved by a *SP_SAVE_PRED* instruction that has already been created earlier. Then, if a $false$ block exists, the instructions from the $false$ block are appended to the branching block. The now empty $false$ block is then removed. Subsequently the same is done for the *true* block.

Note that the appending order of the *true* and $false$ block is completely arbitrary until one of the blocks manipulates the guarding variable which is referenced within the other. In the current implementation the previous transformations are expecting exactly the order described here.

The example in Figure 4.15a shows a CFG before the transformation, Figure 4.15b depicts the transformation result.



(a) CFG before branch removal

(b) CFG after branch removal

Figure 4.15: Removal of the branching instruction during post-RA processing

Expand Pseudo Instructions

In this pass the pseudo instructions that have previously been inserted are now replaced by target specific machine code. The machine code generated in this pass has to implement the functions described in Section 4.5.5. The following paragraphs describe the implementation for each of the pseudo instructions shown there.

<dst> = SP_SAVE_PRED <pred> Sets all bits in the register *dst* to 1 when the predicate *pred* is met by the current APSR, to 0 otherwise. All bits are set to simplify the implementation of a possibly following *SP_SAVE_PRED_COMBINED* instruction.

The following listing exemplary shows the expansion of a *SP_SAVE_PRED* instruction.

```
1  # implements r1 = SP_SAVE_PRED gt
2    mov r1, #0
3    mvngt r1, #0
```

Listing 4.10: SP_SAVE_PRED expanded

<dst> = SP_SAVE_PRED_COMBINED <predreg> <pred> Combines the current APSR with a previous saved predicate. When the current APSR meets the predicate *pred*, the bit representing *pred* is cleared in *predreg* and the result assigned to the register *dst*. The bit representing a certain predicate is determined by the index of the predicate in Table 3-4, Condition code suffixes of [7], with the first entry being the second least significant bit. The first bit retains the original predicate value and is never overwritten.

The following listing exemplarily shows an expansion of an *SP_SAVE_PRED_COMBINED* instruction. Note that *lt* is the 12th entry in Table 3-4 Condition code suffixes of [7] and that the condition-code *ge* is the inverse of *lt*.

```
1  # implements r1 = SP_SAVE_PRED_COMBINED r1 lt
2    bicge r1, #12
```

Listing 4.11: SP_SAVE_PRED_COMBINED expanded

SP_EVAL_PRED <predreg> <restore_carry> <pred> Restores the value of the APSR based on the information in *predreg*. When the predicate *pred* is passed to this instruction the appropriate bit in *predreg*, containing the combined predicate information of a previous *SP_SAVE_PRED_COMBINED* instruction, is evaluated. Without *pred* the least significant bit is evaluated. When the evaluated bit is set, indicating the condition would have been met by the original condition flags the current condition is changed to *ne*. Otherwise, if the original condition flags would not have met *pred*, the condition flags are changed to *eq*.

The following listing exemplarily shows an expansion of an *SP_EVAL_PRED* instruction. It evaluates the condition saved into register *r1* as shown in the example in Listing 4.11.

```
# implements r1 = SP_EVAL_PRED lt
  tst r1, #12
```

Listing 4.12: SP_EVAL_PRED expanded

Predicate Instructions

In the final step of the transformation the instructions of the original *true* and *false* blocks are marked for conditional execution. The process of applying an execution condition to an instruction is called predication here.

The predication is performed in instruction order, starting at the first instruction inherited from the *false* block.

As long as the condition flags in APSR contain the state that would have resulted from the original branching condition, the instructions in the *true* block are predicated with the branching condition, the instructions in the *false* block with the inverse of the branching condition.

When the condition flags in APSR become clobbered or when an already predicated instruction is encountered the condition flags in the APSR cannot directly be used as predicates. They either need to be restored or would be required to carry a combination of two predicates. This is implemented by the pseudo instructions *SP_SAVE_PRED*, *SP_SAVE_PRED_COMBINED* and *SP_EVAL_PRED* which previously have been carefully crafted to set the condition flags in APSR to a state representing the required condition. The following instructions are then predicated accordingly. For the implementation of the *SP_EVAL_PRED* pseudo instruction as shown in Listing 4.12, this means applying the execution condition *ne* to any following instructions.

CHAPTER 5

Results

Contents

5.1 Runtime **101**
5.1.1 Binary search 102
5.1.2 SolveCubic 105
5.2 Cost Drivers **109**
5.2.1 Predicate Calculation 109
5.2.2 Iteration Bounds 110
5.2.3 Nesting 111

5.1 Runtime

In the following the runtime behavior of programs before and after SP-transformation is evaluated.

Experiments regarding the execution time behavior of several basic algorithms have been conducted in [34]. In this work the same algorithms have had the automatic single-path conversion applied to, examining the differences in code size, memory usage and execution timing of the resulting binaries.

The experiments described here have been conducted using a Stellaris EDS-LM3S8962 [40] evaluation kit from Texas Instruments. The CPU[1] is configured to use an 8 MHz clock signal. The clock is driven directly from the on-board oscillator, bypassing all clock dividers and PLL[2]s.

Measurements are done by using the integrated Timer. The timer is configured to use the system clock as timer source. Dividers are disabled, so the timer does count clock cycles.

[1] Central Processing Unit
[2] Phase-Locked Loop

5.1.1 Binary search

The code for this experiment was taken from [34], "Figure 1. Traditional Binary Search." and "Figure 2. WCET-Oriented Version of Binary Search.". To enable automatic generation of single-path code, the loop was annotated with a loop-iteration bound of 5. The loop-iteration bound can be specified as a fixed value here because the experiment will execute search operations only in arrays consisting of exactly 16 elements.

Algorithms analyzed:

- **binsearch_avg** A conventional implementation of the binary-search algorithm, optimized for the average case. Source code is provided in Listing A.4.
- **binsearch_wcet** An implementation of binary-search, manually optimized for low jitter by using the C conditional-expression operator. Source code is provided in Listing A.5.

Build types compared Three different builds have been compared. These are called SP build, SP-prepared build and regular build. A short description of each build type follows.

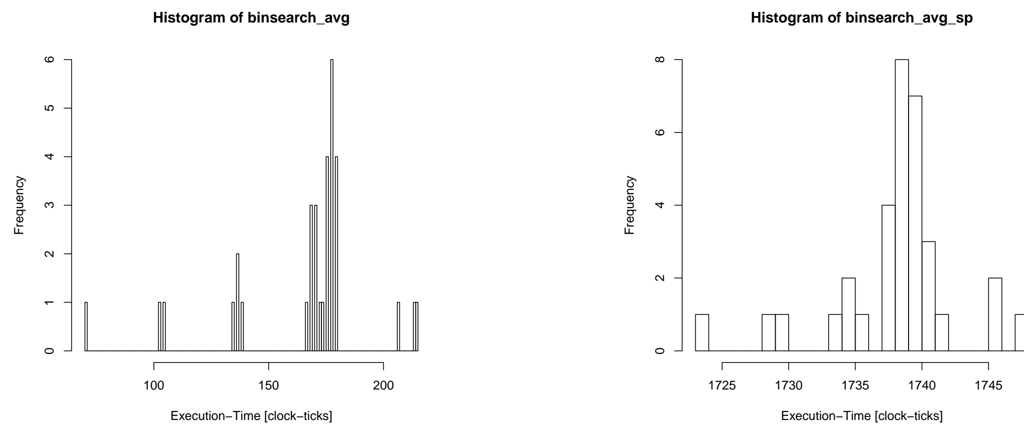
- **SP** This build is a full SP-transformation of the program.
- **SP-prepared** This build does the complete single-path transformation, but without the final step, the if-conversion. The loop conversion is applied, so the number of loop iterations is input independent. Also all nested conditions are transformed into simple conditions that could be directly used to do the if-conversion. All the code the single path transformation will ultimately create is in place, but the branches are not eliminated. The resulting program does still skip conditionally executed code portions instead of executing them under a predicate.
- **Regular** This build is optimized towards low execution times. It does not take any measures to generate code that yields constant execution time.

Runtime Comparison

The following tables compare the runtime of calls to *binsearch_avg* and *binsearch_wcet* for different compilation options.
The execution-time histogram for the SP-prepared build of *binsearch_avg* is given in Figure A.4, for *binsearch_wcet* in Figure A.5.

	Regular	SP
Code-Size [bytes]	84	770
# Cond. Branch-Instructions	5	1
Min. execution-time [clock-cycles]	71	1724
Avg. execution-time [clock-cycles]	166.3	1738.5
Max. execution-time [clock-cycles]	215	1748
Jitter [clock-cycles]	144	24
Execution-Time Distribution	see Figure 5.1a	see Figure 5.1b

Table 5.1: Runtime Comparison of calls to binsearch_avg



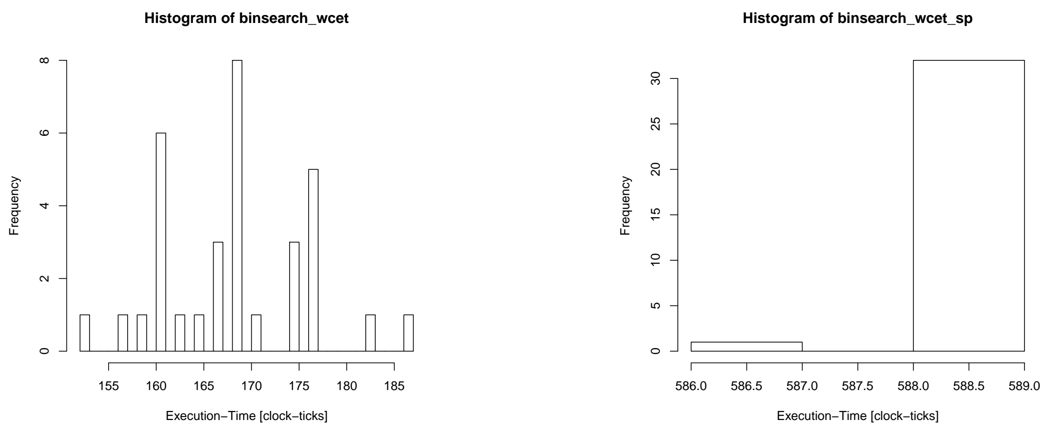
(a) Regular build. For a detailed graph please refer to Figure A.6

(b) SP build. For a detailed graph please refer to Figure A.2

Figure 5.1: Execution time distributions of calls to binsearch_avg

	Regular	SP
Code-Size [bytes]	62	236
# Cond. Branch-Instructions	3	1
Min. execution-time [clock-cycles]	153	587
Avg. execution-time [clock-cycles]	168.7	588.9
Max. execution-time [clock-cycles]	187	589
Jitter [clock-cycles]	34	2
Execution-Time Distribution	see Figure 5.2a	see Figure 5.2b

Table 5.2: Runtime Comparison of calls to binsearch_wcet



(a) Regular build. For a detailed graph please refer to Figure A.7

(b) SP build. For a detailed graph please refer to Figure A.3

Figure 5.2: Execution time distributions of calls to binsearch_wcet

Detailed analysis

As can be seen in Table 5.1, even a relatively small example as the *binary-search* algorithm shows a large increase in execution time after application of the automatic single-path conversion. The maximum observed execution time for the conventionally compiled *binary-search* example has been 215 clock cycles, whereby the single-path converted variant has shown a maximum execution time of 1748 clock cycles. This is a more than 8-fold increase in execution time. To find the cause for this large increase in execution time, the generated programs have been compared on the machine instruction level. Listing A.6 shows the assembler instructions resulting from a compilation without SP-transformation, Listing A.7 shows the resulting assembler code with an application of the SP-transformation. The instructions in these listings have been colored to reflect their categories according to the following list.

The machine instructions of the inner loop for the *binary-search* program have been

categorized into the following four different types.

Program Instructions that constitute the programs functionality.

Spill Instructions that spill register values onto the stack or reload previously spilled values.

Move Instructions that move values between registers to prevent spilling the register values onto the stack.

Guarding Instructions that have been introduced by the SP-transformation to calculate guarding conditions.

In Table 5.3 the result of the analysis is summarized. As one can see there, the number of instructions has been largely increased because of the additional guard-value calculation instructions. However, the currently generated guarding code is highly inefficient. These calculations could be reduced to a large extent by optimizing them. This can be seen in detail in Listing A.7. Also note that the number of program instructions has been slightly reduced. This is because some of the original branching instructions do no longer exist after the SP-transformation.

The overall number of instructions in this example has increased by a factor of approximately 7.7, which is, in this example, close to the factor of 8.1 by which the runtime has increased. This is because the *binary-search* program examined here executes most of the instructions contained within the inner loop on every loop iteration.

The execution-time jitter remaining in the SP-transformed program results from load and store instructions since these are the only instructions which may not execute in exactly one clock cycle. This can also be seen in the assembler code provided in Listing A.7. For detailed information on the timing of the individual instructions on the Cortex-M3 architecture please refer to [8], page 3-4ff.

	Regular	SP
Program [# of instructions]	34	29
Spill [# of instructions]	0	36
Move [# of instructions]	3	14
Guarding [# of instructions]	0	208
Overall [# of instructions]	37	287

Table 5.3: Instruction classification of the inner loop in binsearch_avg

5.1.2 SolveCubic

The following experiments have been conducted using the MiBench [12] benchmark suite, version 1.0. MiBench is a collection of different benchmarks, which are grouped into 6 categories. In the following the focus will be on the *basicmath* test, which is part of the automotive test group. The experiment is additionally restricted to the *basicmath_small* variant due to memory constraints of the test platform. The test platform is based on a Stellaris-LM3S8962 [41] microcontroller which provides 64 KB SRAM and 256 KB FLASH.

The execution behavior of the function *SolveCubic*, which forms a large portion of the *basicmath_small* test, is analyzed in detail here. *SolveCubic* implements a cubic function solver. The following execution times measured are those of a single call to *SolveCubic*, solving equation 5.1. The static call graph of the compiled function is shown in Figure A.1. The static call graph is the same for the single-path variant of the program and the conventionally compiled one. The measured execution times for this single invocation of *SolveCubic* are listed in Table 5.4.

$$1.0 * x^3 - 10.5 * x^2 + 32 * x - 30 = 0 \tag{5.1}$$

Execution Time Measurement

Table 5.4 lists execution times for the three different compilations of the same program.

Regular The first compilation, denoted as Regular, is an ordinary, execution-time optimized compilation similar to the compilation result of a stock LLVM version.

SP-prepared The result set named SP-prepared was obtained by performing all preparing steps of the SP-transformation, but without the final step of replacing the branches by predicated instructions. Function calls are performed as in the SP-converted variant of the program, i.e., also functions that would not be executed in the original program are called but their effects are isolated by passing an appropriate guarding value. The notable exception are calls to functions in Compiler-RT. Since these are introduced late in the compilation process the transformation of the program structure does not consider them. This is done during the predication phase. Because the function calls may be skipped in the SP-prepared variant whereby they are always executed with the fully SP-transformed variant, the number of calls for functions in Compiler-RT increases from the SP-prepared variant to the SP variant.

SP Finally there is a set of execution-time measurements for the fully SP-transformed program, denoted SP. In contrast to the SP-prepared variant input dependent branches in the program are replaced by predicated instructions.

For each of the compilation types the execution time has been measured. This measurement has been done by setting a hardware timer to run synchronous to the CPU clock, essentially counting clock cycles. All time measurement results are therefore in clock ticks. The clock was set to 8.0 MHz, derived directly from the 8.0 MHz crystal provided by the Stellaris LM3S8962 Evaluation Board [40].

Execution Time

The following paragraphs describe the contents of the individual columns in Table 5.4.

Name Name of the function whose execution time is measured.

Time Execution time, in clock cycles, for this function including all functions called. All functions were instrumented to compute the differences of the timer value between function entry and function exit, yielding the runtime in clock cycles. This runtime has been accumulated for multiple executions.

Number of invocations of the current function.

RT-factor Runtime factor of the time spent executing the current function and its sub-functions compared to the regular compilation.

Vertically the results are split into 3 parts to illustrate the different program parts. These 3 layers are, from top to bottom, MiBench, libmusl and Compiler-RT. The fact that the program is organized into these 3 layers is also reflected in the call graph shown in Figure A.1.

Interpretation

For this particular invocation, the single-path compiled variant of *SolveCubic* requires the 1473-fold time to execute than the conventionally compiled variant does. The much smaller *binary-search* example presented in Section 5.1.1 in contrast has only shown a 10- to 20-fold increase in execution time.

The data from Table 5.4 allows to roughly distinguish between two causes for this increase in runtime. Roughly refers to the fact that SP-prepared does not include all calls to Compiler-RT functions, as described above. The two causes are the control-flow modifications with predicate calculations and the "execution" of disabled code parts. The former is included in the times provided by the SP-prepared columns, the latter in the SP columns.

The largest portion of execution-time growth is caused by the iteration over disabled code parts, as can be determined by comparing the times provided in the columns SP-prepared and SP. In this example it is causing approximately a 73-fold increase in execution time. That means that, on the now unique instruction trace, only $1/73$ of the instructions are required to calculate the result for that particular invocation.

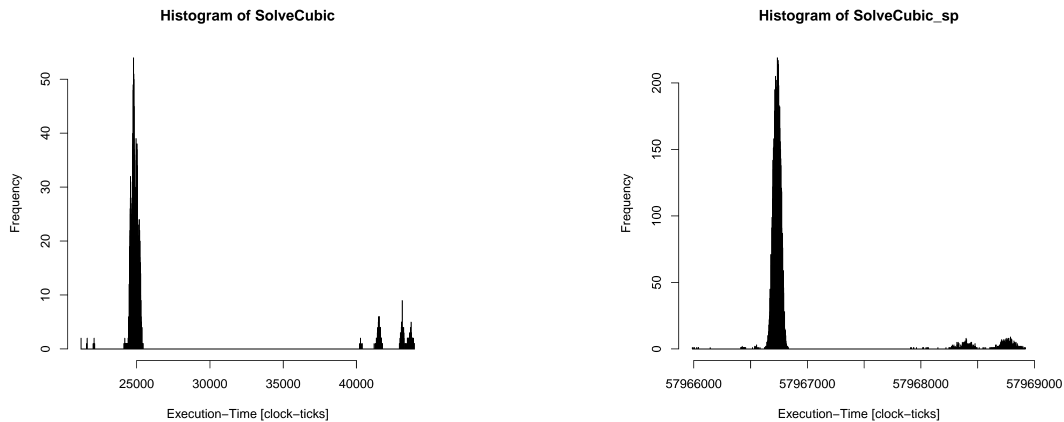
	Regular		SP-prepared			SP		
Function	Time	#	Time	#	RT-factor	Time	#	RT-factor
SolveCubic	39351	1	793978	1	20	57968641	1	1473
cos	6494	1	38341	1	6	162703	1	25
atan	8234	2	48476	2	6	88498	2	11
cos	15134	3	633691	3	42	57348569	3	3789
floor	0	0	0	0		497520	60	
pow	0	0	7247	1		223950	1	
scalbn	0	0	17219	67		1101344	127	
sqrt	5885	5	48913	8	8	124406	8	21
__rem_pio2	2627	3	578138	3	220	56861057	3	21645
__rem_pio2_large	0	0	564561	3		56565732	3	
_adddf3	10283	93	79764	95	8	29213803	18908	2841
__cos	0	0	0	0		0	0	
__divdf3	2159	13	9199	13	4	37855	33	18
__eqdf2	0	0	0	0		23298	66	
__fixdfsi	0	0	0	0		81702	801	
__floatsidf	0	0	0	0		322960	3670	
__gtdf2	66	2	197	1	3	64600	183	979
__ledf2	0	0	156	1		314	1	
__ltdf2	0	0	0	0		22592	64	
__muldf3	9239	96	70295	97	8	23888715	18943	2586
__sin	0	0	0	0		0	0	
__subdf3	4763	25	22883	25	5	1167529	725	245
__unorddf2	4	1	0	0	0	5632	64	1408

Table 5.4: Execution times for a call to SolveCubic with different compilation types

Runtime Comparison The following table compares the runtime of all calls to *SolveCubic* during the execution of the *basicmath_small* benchmark for different compilation options.

	Regular	SP
Code-Size [bytes]	25210	77814
# Cond. Branch-Instructions	426	110
Min. execution-time [clock-cycles]	21203	57965986
Avg. execution-time [clock-cycles]	25851	57966836
Max. execution-time [clock-cycles]	43953	57968921
Jitter [clock-cycles]	22750	2935
Execution-Time Distribution	see Figure 5.3a	see Figure 5.3b

Table 5.5: Runtime Comparison of all calls to SolveCubic in basicmath_small



(a) Regular build. For a detailed graph please refer to Figure A.8.

(b) SP-transformed build. For a detailed graph please refer to Figure A.9

Figure 5.3: Execution-Time distribution of calls to SolveCubic

5.2 Cost Drivers

Timing measurements done in Section 5.1 have shown a large increase in execution time. In this section some reasoning about the causes for this execution-time increase is done, especially on the effect the program size has on the execution time.

5.2.1 Predicate Calculation

The predicate calculation is done for every basic block and its runtime cost may be estimated to be in $O(\text{basic-block size})$ for each particular block, since each predicate

use requires some constant-time processing. Blocks with more instructions have more opportunities to make use of predicates. Once for each instruction.

In the SP-transformed program variant each instruction will be executed. For each instruction a certain processing overhead is generated that does not grow with program size. These predicate calculations are always executed before the actual program instructions are. Therefore the predicate handling is causing some constant execution-time factor for SP-transformed programs.

In the current implementation a rough upper bound for the execution-time increase may be deducted from the increase of the executable code size that is caused by the SP-transformation. The reason therefore is that on the Cortex-M3 target all instructions are similar in size and the predicate calculation is only using simple instructions that are usually executed within one clock cycle. Any growth in executable size which is caused by the predicate calculation will therefore lead to an increase in execution time due to the necessity to execute this new instructions. Should the previously existing code also consist of simple instructions that execute within a single clock cycle, the factor by which the runtime increases from the new predicate-calculating instructions can be assumed to be close to factor in which the code size increased. Actual programs will also consist of, in terms of clock cycles required for their execution, more expensive instructions, so the relative execution-time growth will be below the relative code size growth.

The results in Section 5.1 show that SP-transformation increases code size by a factor between 2 and 8 times, depending on the program transformed. In these cases optimizations to the predicate calculation may at most reduce execution time to $\frac{1}{8}th$.

5.2.2 Iteration Bounds

In the SP-transformed program each input-data dependent loop is always executed for the number of iterations that has been annotated. When the iteration bound provided for a certain loop is not higher than the number of iterates executed in its worst-case, for the worst-case there is no additional cost due to additional loop iterations in the SP-program compared to the conventionally compiled program.

Additional runtime is required when the specified iteration bound IT_{annot} is larger than the number of loop iterations executed in the wost-case IT_{WC}. The number of additional loop iterations that the SP-program executes compared to the worst-case execution of the conventional program may be calculated by: $IT_{SP} = IT_{annot} - IT_{WC}$. Assuming that the time required for the execution of a single loop iteration is T_{IT}, the additional execution-time cost due to the larger than necessary iteration bound is:

$$T_{SP} = IT_{SP} * T_{IT} \tag{5.2}$$

When loops are nested, the execution time of an outer loop contains the execution time of any nested loops. Let $Tout_{IT}$ be the average iteration time of the outer loop and Tin_{IT} the execution time of the inner loop. The iteration time of the outer loop is a combination of the time spent in the inner loop and the time T_x spent execution loop

content not part of the inner loop. $ITin_{SP}$ denotes the additional iterations, caused by a larger than necessary iteration-bound annotation, of the inner loop.

$$Tout_{IT} = T_x + (ITin_{WC} + ITin_{SP}) * Tin_{IT} \tag{5.3}$$

When the outer loop's iteration annotation exceeds the worst-case iteration bound $ITout_{WC}$ by $ITout_{SP}$, the complete execution time for an invocation of the outer loop may be calculated as follows.

$$\begin{aligned} Tout &= ITout_{annot} * Tout_{IT} && (5.4)\\ &= (ITout_{WC} + ITout_{SP}) * Tout_{IT} && (5.5)\\ &= (ITout_{WC} + ITout_{SP}) * (T_x + (ITin_{WC} + ITin_{SP}) * Tin_{IT}) && (5.6)\\ &= ITout_{WC} * ITin_{WC} * Tin_{IT} + \dots && (5.7) \end{aligned}$$

So, for two nested loops with larger than necessary iteration bounds the execution-time growth is in $o(ITout_{WC} * ITin_{WC})$. Any further nesting in a loop with a larger than necessary iteration bound will further increase the execution time in the same way.

In general, for n nested loops with imprecise iteration bounds, where each iteration bound is at least $ITmin_{SP}$ too large, the growth of execution time will be in:

$$o(ITmin_{SP}{}^n) \tag{5.8}$$

To reduce the effect of this source for execution-time growth keep the number of loop-nesting levels low and use precise loop-iteration bounds.

5.2.3 Nesting

The number of nesting levels of SESE-regions also has implications on the execution time of the SP-transformed program. For simplicity, in the following all branches are assumed to be input-data dependent. Non input-dependent branches starting SESE-regions do not increase execution time, since they are preserved in the transformed program. Loops are ignored here, since the effect to loop iterations is already considered in the previous section, and anything said in this section should also work for programs that have each loop completely unrolled.

Any SESE-region contains at least one basic block, which is not necessarily executed, for a given invocation of the region entry and exit. So, when the SESE-region entry is passed, there is a probability p that the block contained in the region is executed before the region exit is passed. The probability for not executing this block is $1 - p$.

Nesting the SESE-region inside another SESE-region adds a probability p_1 that the inner region is not executed at all when the outer regions entry block is passed. When the outer region is entered, the probability for the block contained within the inner region to be executed before the outer regions exit block is passed is $p_1 * p$. Adding more levels of nesting on average decreases the probability for all contained blocks, be it directly or

in subregions, to be executed. Assuming a uniform execution time t for all blocks, on average the entire region executes in time T:

$$T = t + t * p + t * p_1 * p \tag{5.9}$$

Call Graph Anything said above for SESE-regions is also true for function calls. So one has to regard the number inter-procedural nesting levels here.

Since a program does not only grow by adding new nested regions, a region may also contain a sequence of subregions. Let n_l be the number of basic blocks on nesting level l and further unifying all execution probabilities to $1/p$, the average execution T time for 2 nesting levels is:

$$T = n_0 * t + n_1 * t * p + n_2 * t * p_1 * p \tag{5.10}$$

In general for L nesting levels:

$$T = \sum_{l=0}^{L-1} n_l * t * p^l \tag{5.11}$$

Further assuming a uniform distribution of the n basic blocks amongst nesting levels, resulting in n/L basic-blocks per level, yields

$$T = \sum_{l=0}^{L-1} \frac{n}{L} * t * p^l \tag{5.12}$$

$$= \frac{n}{L} * t * \sum_{l=0}^{L-1} p^l \tag{5.13}$$

Effect on SP-programs In the single-path variant each basic block is executed once. Resulting in $T_{SP} = n * t$. The runtime factor from SP-conversion is therefore:

$$\frac{T_{SP}}{T} = L * \frac{p-1}{p^L - 1} \tag{5.14}$$

Since the probability $p < 1$, so p^L decreases for larger values of L. The runtime factor of the single-path transformed program in relation to the conventional compiled program may, by all the assumptions made above, be expected to linearly grow by the number of nested regions in the program. For real programs at least the uniform distribution of basic blocks amongst nesting levels may not be true. Should the number of blocks decrease towards in the inner nesting levels, the runtime factor will be lowered.

Relationship of Nesting-Depth to Program Size Empirical studies have shown that the nesting depth within a procedure does not increase with the procedure size [19]. Therefore growth in the inter-procedural nesting depth will most likely be the result of an increased call-graph depth.

CHAPTER 6

Conclusion and Outlook

Conclusion The implementation of the SP-transformation done in the course of this thesis has shown that it is feasible to automatically generate an SP-program from the annotated sources of a high-level programming language.

In Section 2.3 a transformation scheme for any program that may be represented as a reducible flow-graph and has iteration bounds given into a single-path program has been presented. This transformation scheme corresponds to the SP-transformation rules publicized earlier which where specified for an imperative programming language. The specification at control-flow level has the advantage to not require a mapping from the concrete program-implementation language to the SP-transformation rules. Instead the transformation may be applied at the control-flow level, whereby the transformation into a control-flow representation is likely to have been already implemented with the compilation process anyway.

Experiments with programs resulting from the automatic SP-transformation have shown that the converted programs may require a much longer execution time than the unconverted program's WCET. Whereby this increase in runtime depends on the structure of the converted program. The impact of input data on the programs execution time has been reduced by the SP-transformation. It has, however, not completely disappeared, since the transformed programs make use of the conditional-execution mechanism provided by the target hardware. This conditional execution does not have the property of a constant execution time as it is provided by the constant-time conditional expression that has been used when specifying the SP-transformation. Therefore the execution times still vary with the input data provided.

Outlook It remains to be analyzed which algorithms are suited best for an automatic conversion into an SP-program. It might be that these are commonly programs that solve tasks that are typical for real-time applications.

Further, another implementation of the constant-time conditional expression may be used with the automatic SP-transformation to generate programs that show even less execution time jitter than ones generated by the current implementation.

The execution of the transformed programs does currently show a unique sequence of instructions executed. To achieve truly constant execution times on a cached computing architecture further measures for memory accesses have to be provided. One could, for any memory access where the address does not depend on input data extend SP-programs to show a unique pattern on the memory address bus. Since loops in SP-programs always iterate for their worst-case number of times, memory accesses within loops may also be performed when the loop has already logically terminated. To be able to also do this for write accesses the memory location could first be read and, in the disabled case, be re-written with the unmodified data. The resulting programs should not only have a unique addressing pattern for instruction locations but also for data-memory locations.

APPENDIX A

Appendix

A.1 Call Graphs

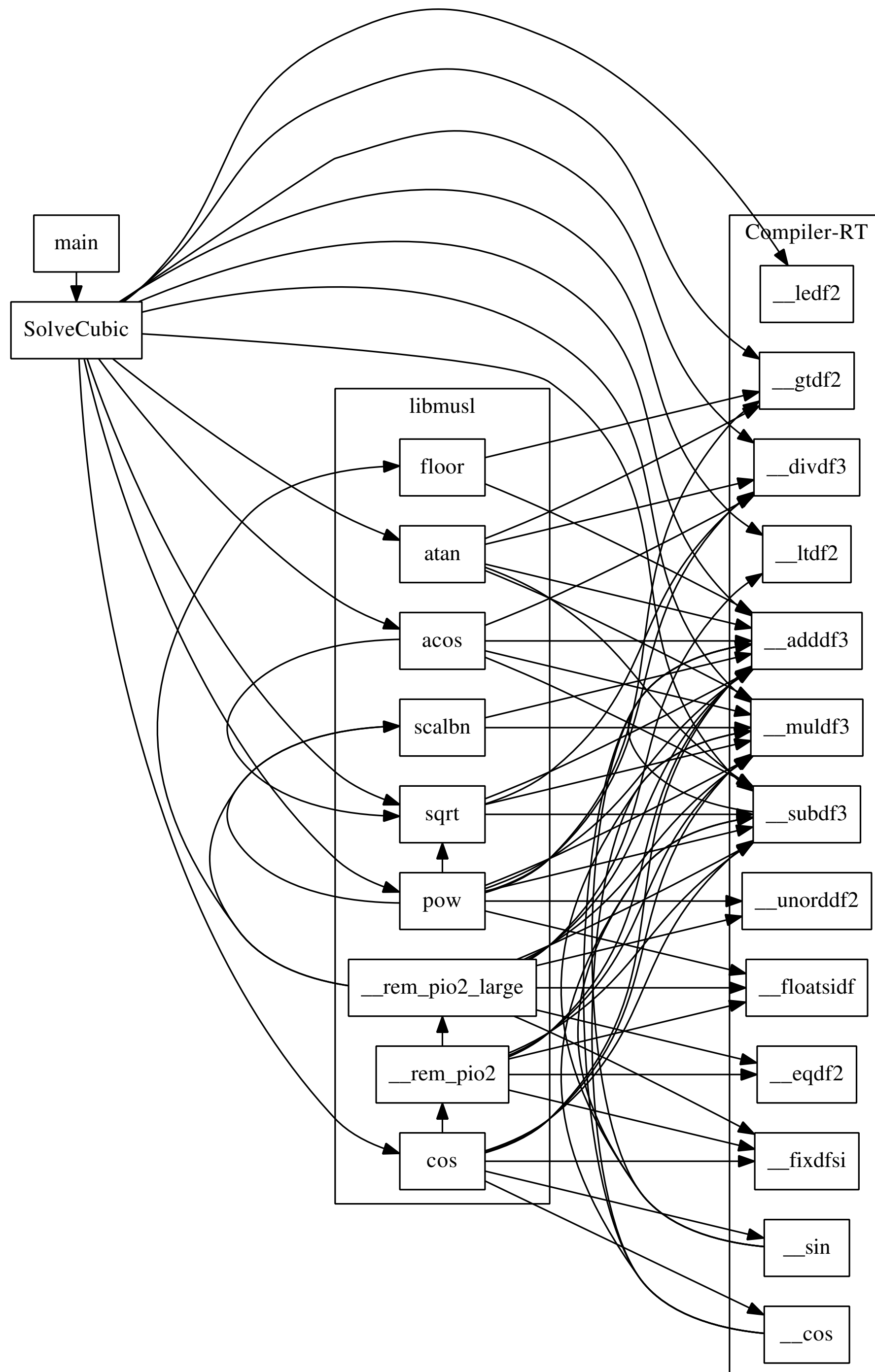


Figure A.1: Static callgraph for SolveCubic

A.2 Execution-Time Distributions

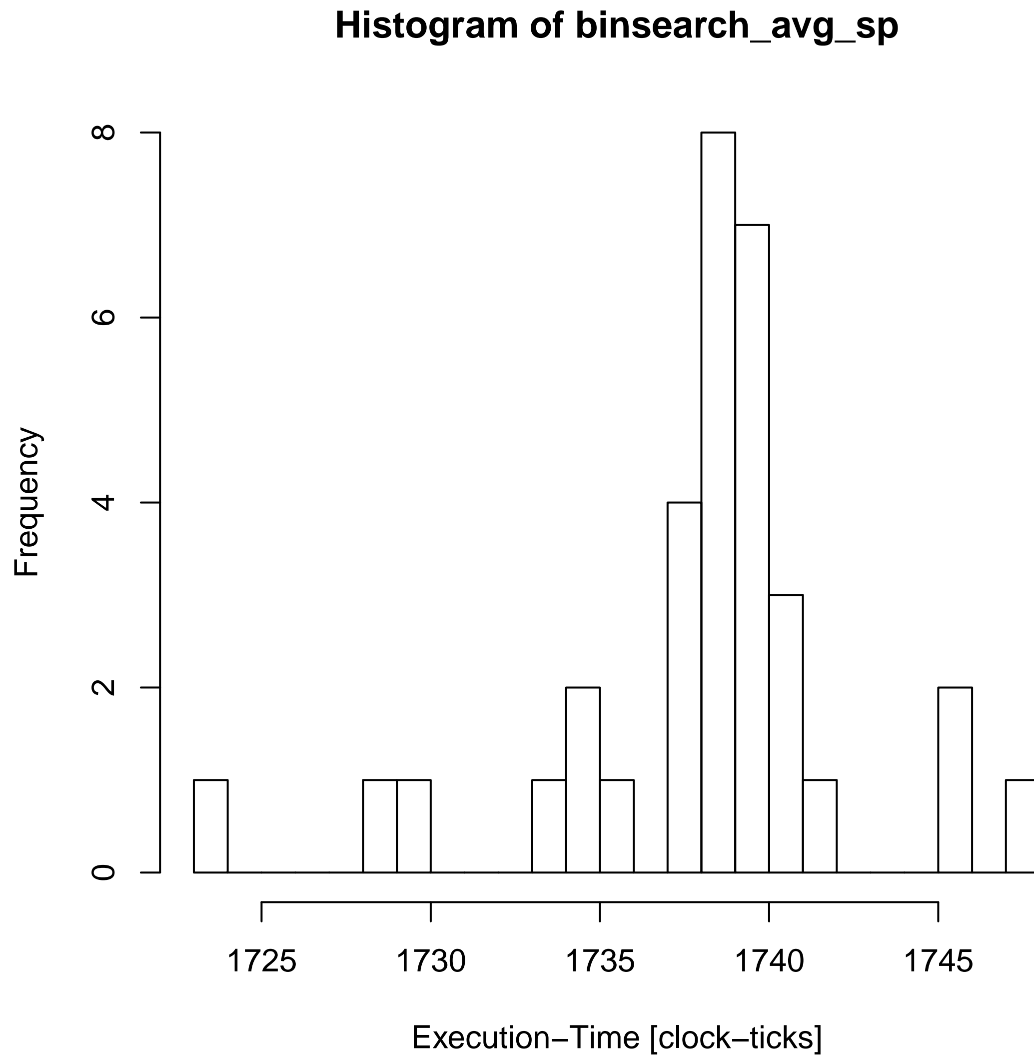


Figure A.2: Execution-Time distribution for calls to binsearch_avg, SP-Build

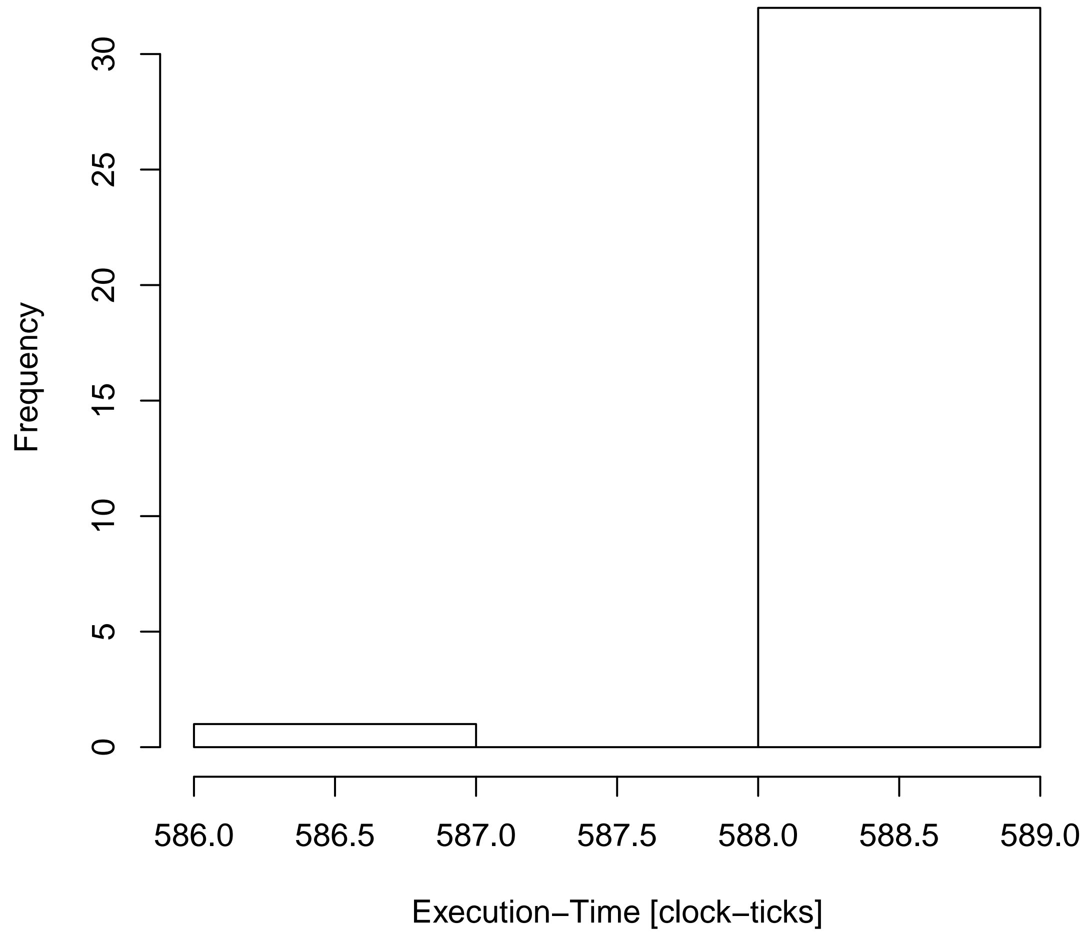


Figure A.3: Execution-Time distribution for calls to binsearch_wcet, SP-Build

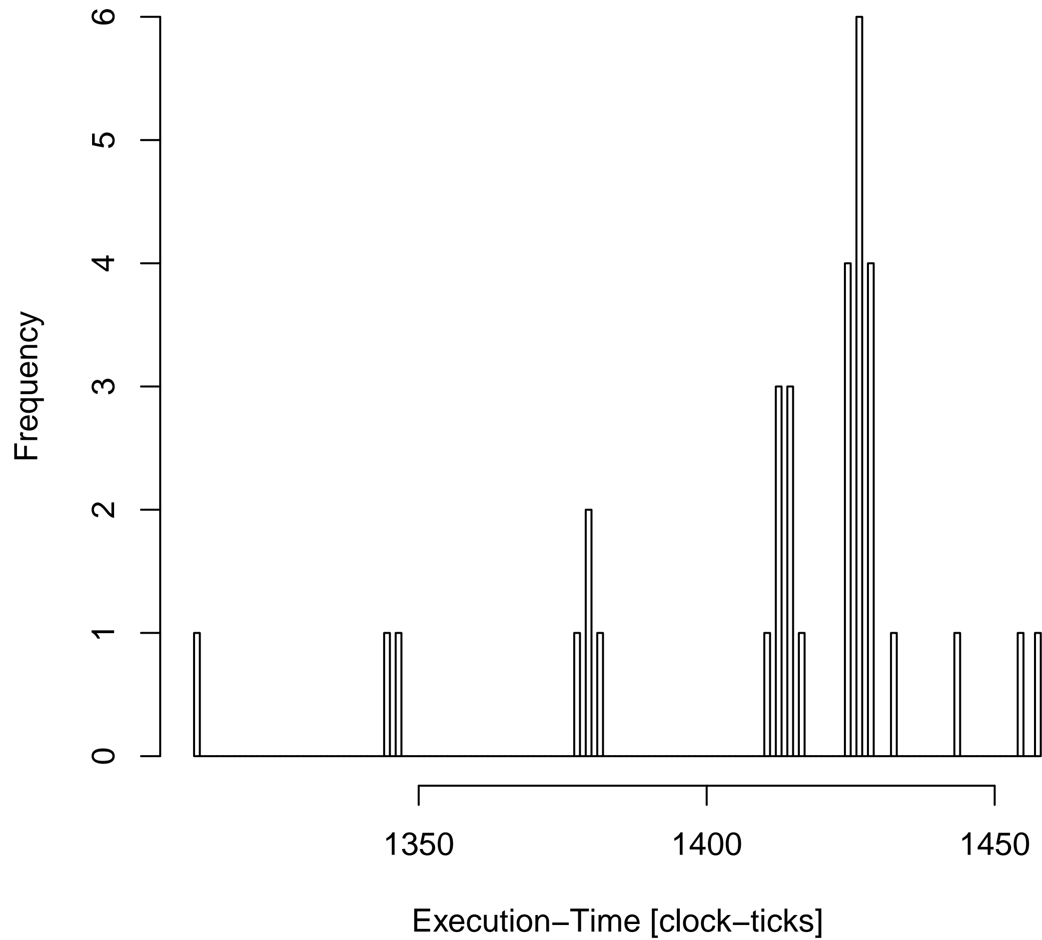


Figure A.4: Execution-Time distribution for calls to binsearch_avg, SP-Prepared

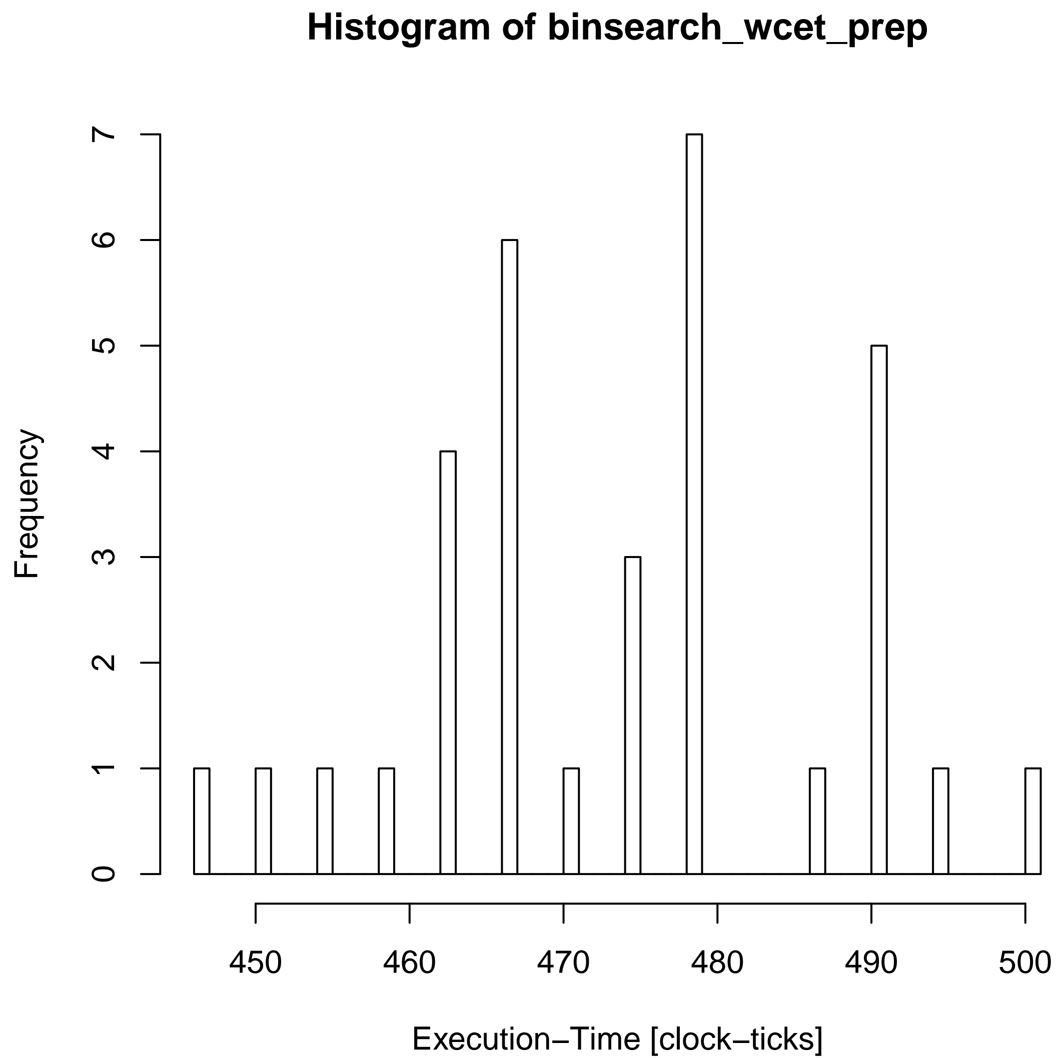


Figure A.5: Execution-Time distribution for calls to binsearch_wcet, SP-Perpared

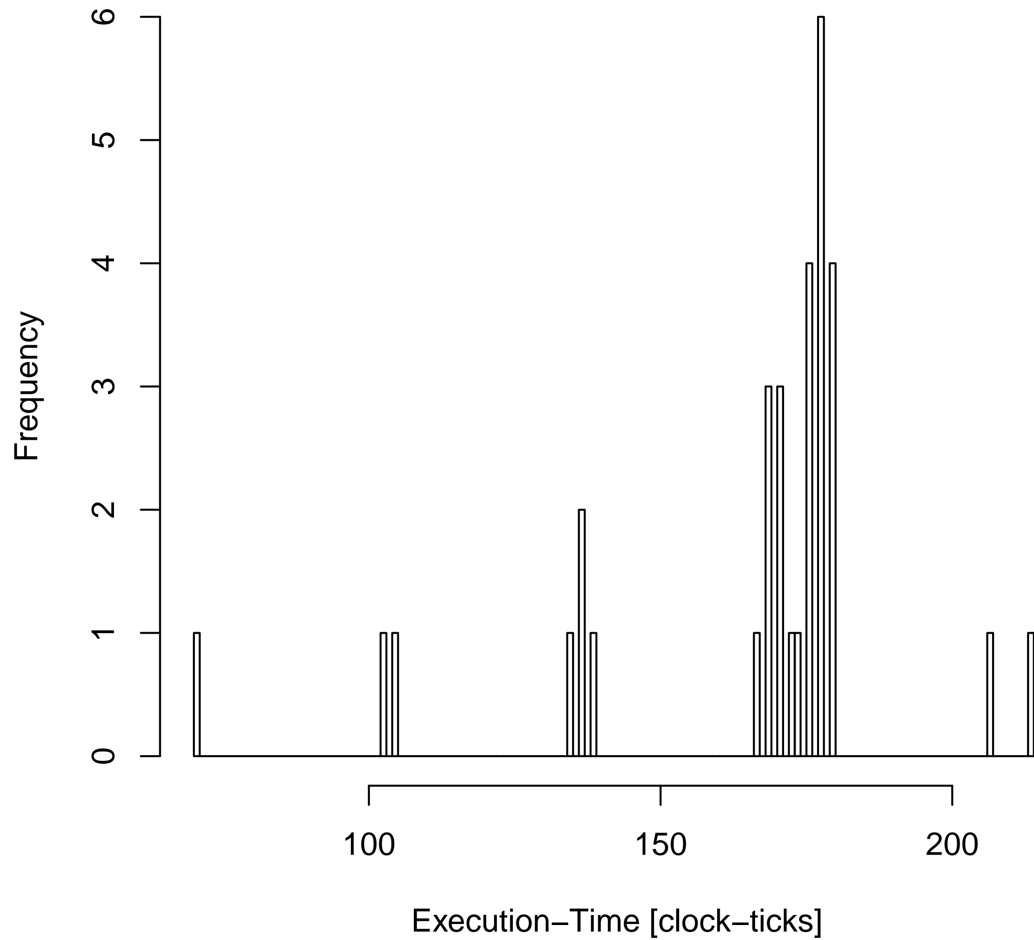


Figure A.6: Execution-Time distribution for calls to binsearch_avg, regular build

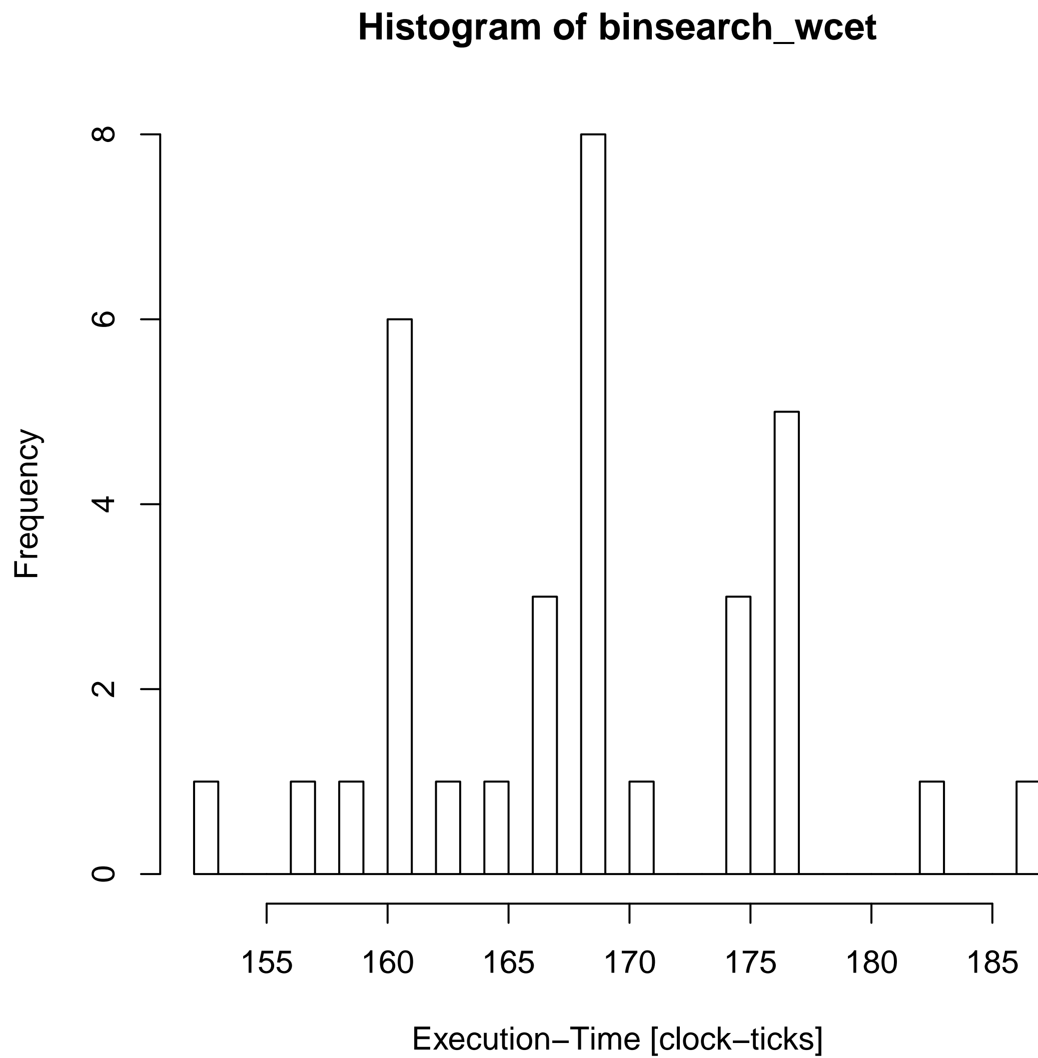


Figure A.7: Execution-Time distribution for calls to binsearch_avg, regular build

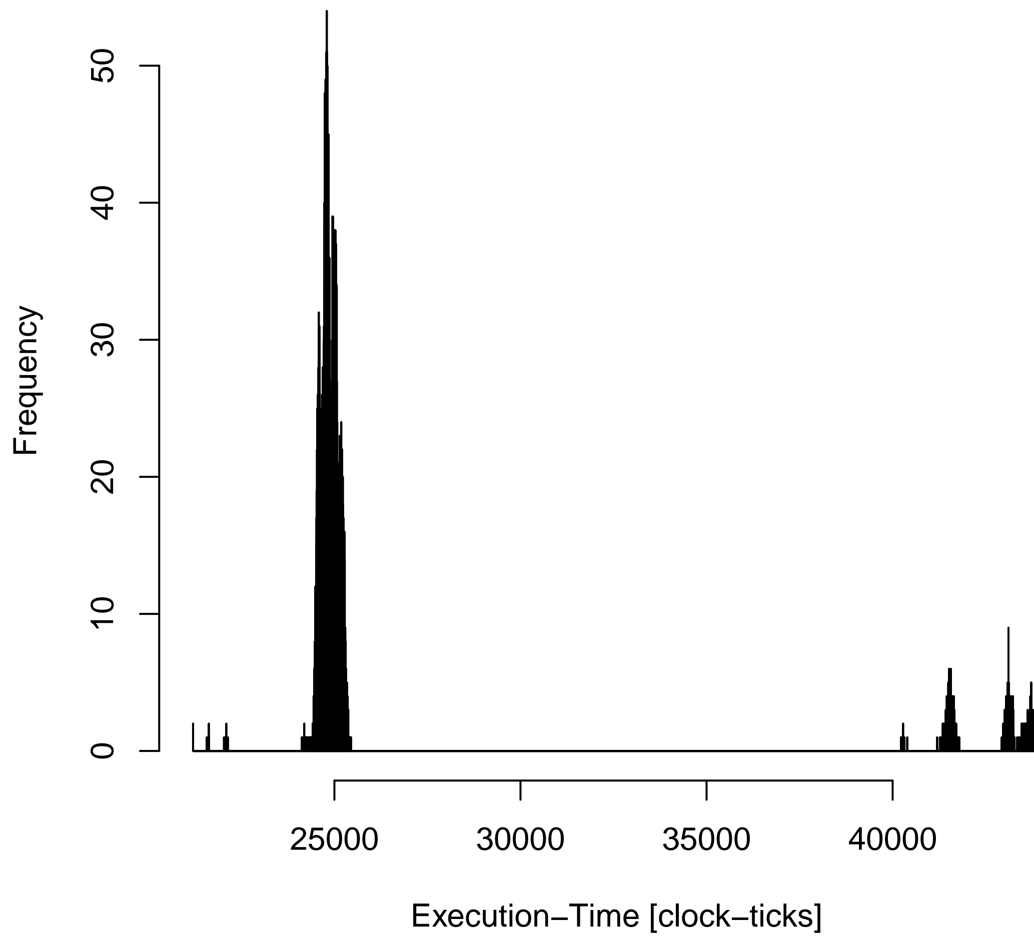


Figure A.8: Execution-Time distribution for calls to SolveCubic

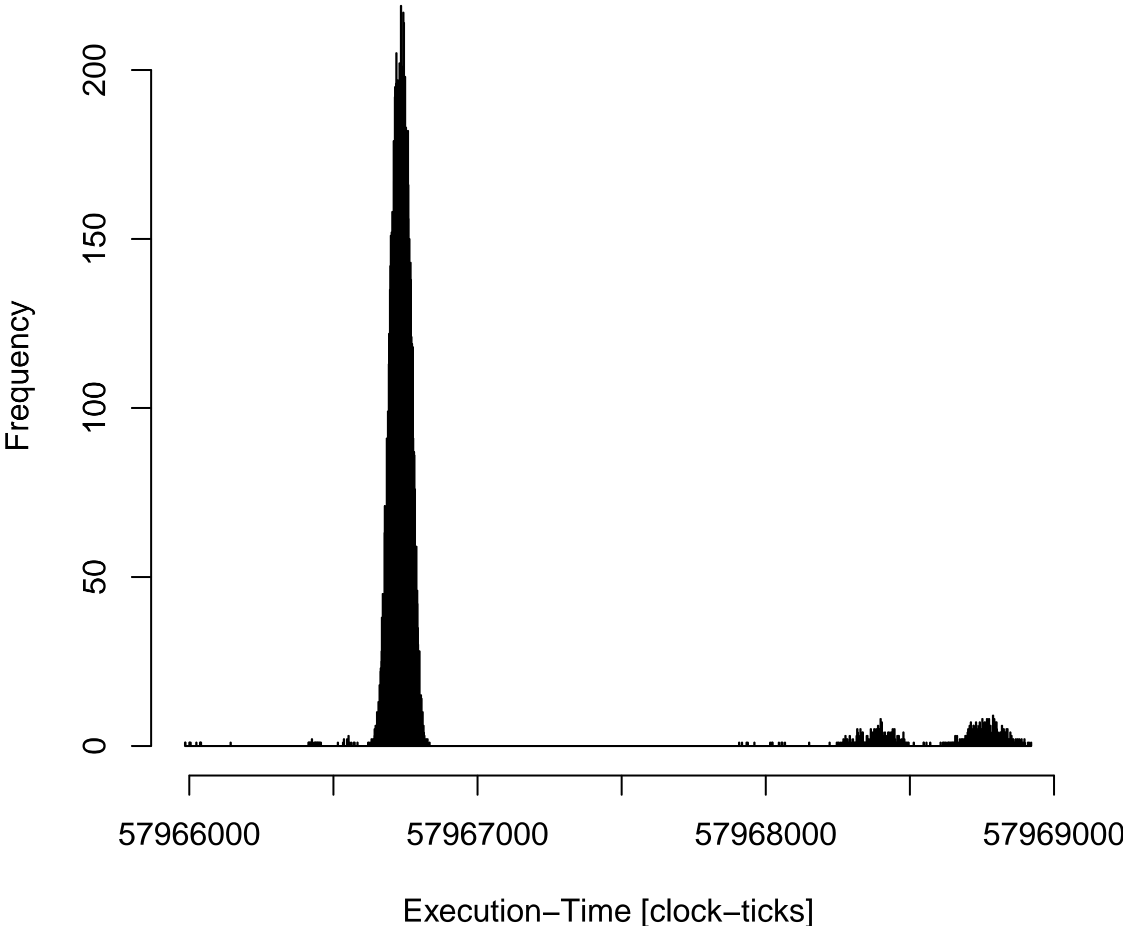


Figure A.9: Execution-Time distribution for calls to SolveCubic, SP-Build

A.3 Listings

```
1  BB#14: derived from LLVM BB %if.then
2      Live Ins: %R4 %R5 %R6 %R8 %R9 %R10
3      Predecessors according to CFG: BB#13
4          SP_EVAL_PRED %R6, pred:14, pred:%CPSR<imp-def,dead>
5          %R0<def> = MOVr %R5, pred:14, pred:%noreg, opt:%noreg;
6          %R1<def> = MOVr %R4, pred:14, pred:%noreg, opt:%noreg;
7          %R2<def> = MOVi 0, pred:14, pred:%noreg, opt:%noreg
8          %R3<def> = MOVi 0, pred:14, pred:%noreg, opt:%noreg
9          BL <es:__nedf2>, %R0<kill>, %R1<kill>, %R2<kill>, %R3<kill>, %R0<
               imp-def>, %R1<imp-def,dead>, %R2<imp-def,dead>, %R3<imp-def,
               dead>, %LR<imp-def,dead>, %CPSR<imp-def,dead>, %SP<imp-use>,
               ...;
10         %R7<def> = MOVr %R0<kill>, pred:14, pred:%noreg, opt:%noreg;
11         CMPri %R7, 0, pred:14, pred:%noreg, %CPSR<imp-def>;
12         %R6<def> = SP_SAVE_PRED_COMBINED %R6<kill>, pred:1, pred:%CPSR<kill
               >;
13         SP_EVAL_PRED %R6, pred:1, pred:%CPSR<imp-def>;
14         %R7<def> = MOVi 1, pred:1, pred:%CPSR, opt:%noreg;
15         SP_EVAL_PRED %R6<kill>, pred:14, pred:%CPSR<imp-def,dead>
16     Successors according to CFG: BB#15
```

Listing A.1: Call to compiler-rt (__nedf2) without additional guard-argument

```
1  BB#14: derived from LLVM BB %if.then
2      Live Ins: %R4 %R5 %R6 %R8 %R9 %R10
3      Predecessors according to CFG: BB#13
4          SP_EVAL_PRED %R6, pred:14, pred:%CPSR<imp-def,dead>
5          %R0<def> = MOVi 0, pred:14, pred:%noreg, opt:%noreg
6          STRi12 %R0<kill>, %SP<kill>, 0, pred:14, pred:%noreg; mem:ST4[Stack
               ]
7          %R0<def> = MOVi 0, pred:14, pred:%noreg, opt:%noreg
8          %R1<def> = MOVr %R5, pred:14, pred:%noreg, opt:%noreg;
9          %R2<def> = MOVr %R4, pred:14, pred:%noreg, opt:%noreg;
10         %R3<def> = MOVi 0, pred:14, pred:%noreg, opt:%noreg
11         BL <es:__nedf2>, %R0<kill>, %R1<kill>, %R2<kill>, %R3<kill>, %R0<
               imp-def>, %R1<imp-def,dead>, %R2<imp-def,dead>, %R3<imp-def,
               dead>, %LR<imp-def,dead>, %CPSR<imp-def,dead>, %SP<imp-use>,
               ...;
12         %R7<def> = MOVr %R0<kill>, pred:14, pred:%noreg, opt:%noreg;
13         CMPri %R7, 0, pred:14, pred:%noreg, %CPSR<imp-def>;
14         %R6<def> = SP_SAVE_PRED_COMBINED %R6<kill>, pred:1, pred:%CPSR<kill
               >;
15         SP_EVAL_PRED %R6, pred:1, pred:%CPSR<imp-def>;
16         %R7<def> = MOVi 1, pred:1, pred:%CPSR, opt:%noreg;
17         SP_EVAL_PRED %R6<kill>, pred:14, pred:%CPSR<imp-def,dead>
18     Successors according to CFG: BB#15
```

Listing A.2: Call to compiler-rt (__nedf2) with const zero as additional guard-argument

```
1  BB#13: derived from LLVM BB %if.then.guard
2      Live Ins: %R4 %R5 %R8 %R9 %R10
3      Predecessors according to CFG: BB#12
4          %R0<def> = LDRi12 %SP, 4, pred:14, pred:%noreg; mem:LD4[%guard]
5          CMPri %R0<kill>, 0, pred:14, pred:%noreg, %CPSR<imp-def>;
6          %R6<def> = SP_SAVE_PRED pred:1, pred:%CPSR
7          SP_EVAL_PRED %R6, pred:14, pred:%CPSR<imp-def,dead>
8          %R0<def> = MOVi 0, pred:0, pred:%CPSR, opt:%noreg
9          STRi12 %R0<kill>, %SP<kill>, 0, pred:0, pred:%CPSR; mem:ST4[Stack]
10         %R0<def> = MOVi 0, pred:0, pred:%CPSR, opt:%noreg
11         %R1<def> = MOVr %R5, pred:0, pred:%CPSR, opt:%noreg;
12         %R2<def> = MOVr %R4, pred:0, pred:%CPSR, opt:%noreg;
13         %R3<def> = MOVi 0, pred:0, pred:%CPSR, opt:%noreg
14         %R6<def> = SP_SAVE_PRED pred:0, pred:%CPSR;
15         %R0<def> = MOVi 1, pred:1, pred:%CPSR, opt:%noreg;
16         BL <es:__nedf2>, %R0<kill>, %R1<kill>, %R2<kill>, %R3<kill>, %R0<
               imp-def>, %R1<imp-def,dead>, %R2<imp-def,dead>, %R3<imp-def,
               dead>, %LR<imp-def,dead>, %CPSR<imp-def,dead>, %SP<imp-use>,
               ...;
17         %R7<def> = MOVr %R0<kill>, pred:14, pred:%noreg, opt:%noreg;
18         CMPri %R7, 0, pred:14, pred:%noreg, %CPSR<imp-def>;
19         %R6<def> = SP_SAVE_PRED_COMBINED %R6<kill>, pred:1, pred:%CPSR<kill
               >;
20         SP_EVAL_PRED %R6, pred:1, pred:%CPSR<imp-def>;
21         %R7<def> = MOVi 1, pred:1, pred:%CPSR, opt:%noreg;
22         SP_EVAL_PRED %R6<kill>, pred:14, pred:%CPSR<imp-def,dead>
23     Successors according to CFG: BB#15
```

Listing A.3: Call to compiler-rt (__nedf2) with additional guard-argument dependent on guarding-value

```
static int binSearch_avg(int key, int a[])
{
        int left = 0, right = SIZE-1, idx, inc;
        int found = 0;
        _LOOP_BOUND(5);
        do
        {
                idx = (right + left) >> 1;
                if (a[idx] == key)
                {
                        found = 1;
                }
                else if (a[idx] < key)
                {
                        left = idx+1;
                }
                else
                {
                        right = idx-1;
                }
        } while (!found && (right >= left));

        if (found)
        {
                return idx;
        }
        else
        {
                return -1;
        }
}
```

Listing A.4: Traditional Binary Search. Taken from [34] page 4, Figure 1. With additional loop-bound annotation.

```
static int binSearch_wcet(int key, int a[])
{
        int left = 0, right = SIZE - 1, idx, inc;

        idx = (right + left) >> 1;

        for (inc = SIZE; inc > 0; inc = inc >> 1)
        {
                right = (key < a[idx] ? idx - 1 : right);
                left = (key > a[idx] ? idx + 1 : left);
                idx = (right + left) >> 1;
        }

        return idx;
}
```

Listing A.5: WCET-Oriented Version of Binary Search. Taken from [34] page 5, Figure 2.

```
binSearch_avg:
        push    {r4, r5, r6, lr}
        mov     r5, r0
        movs    r0, #5
        mov     r6, r1
        bl      _LOOP_BOUND
        movs    r2, #0
        movs    r1, #15
        mov     r3, r2
        b       .LBB5_2
.LBB5_1:
.LBB5_2:
        adds    r0, r1, r2
        asrs    r0, r0, #1
        ldr.w   r4, [r6, r0, lsl #2]
        cmp     r4, r5
        bne     .LBB5_4
        movs    r3, #1
        b       .LBB5_8
.LBB5_4:
        ldr.w   r4, [r6, r0, lsl #2]
        cmp     r4, r5
        bge     .LBB5_6
        adds    r2, r0, #1
        b       .LBB5_7
.LBB5_6:
        subs    r1, r0, #1
.LBB5_7:
.LBB5_8:
        cmp     r3, #0
        bne     .LBB5_11
        cmp     r1, r2
        mov.w   r4, #0
        it      ge
        movge   r4, #1
        b       .LBB5_12
.LBB5_11:
        movs    r4, #0
.LBB5_12:
        cmp     r4, #1
        beq     .LBB5_1
        cmp     r3, #0
        beq     .LBB5_15
        b       .LBB5_16
.LBB5_15:
        mov.w   r0, #-1
.LBB5_16:
        pop     {r4, r5, r6, pc}
```

Listing A.6: Assembler-instructions of the binary-search example given in Listing A.4 when compiled without SP-transformation.

```
1  binSearch_avg:
2          push.w  {r4, r5, r6, r7, r8, r9, r10, r11, lr}
3          sub     sp, #52
4          mov     r4, r2
5          mov     r6, r0
6          movs    r0, #5
7          str     r1, [sp, #36]
8          str     r4, [sp]
9          bl      _LOOP_BOUND
10         cmp     r6, #0
11         mov.w   r10, #0
12         mov.w   r8, #0
13         it      ne
14         addne.w r10, r6, #1
15         str     r1, [sp, #24]
16         movs    r3, #15
17         mvn     r2, #4
18         str     r1, [sp, #32]
19         str     r1, [sp, #28]
20         str     r1, [sp, #20]
21         str     r1, [sp, #48]
22         str     r1, [sp, #44]
23         movs    r1, #0
24         str     r1, [sp, #40]
25 .LBB5_2:
26         cmp.w   r10, #0
27         str     r6, [sp, #4]
28         str     r2, [sp, #12]
29         itt     eq
30         addeq.w r0, r3, r8
31         asreq   r0, r0, #1
32         cmp.w   r10, #0
33         mov.w   r2, #0
34         itttt   eq
35         mvneq   r2, #0
36         ldreq.w r1, [r4, r0, lsl #2]
37         ldreq   r6, [sp, #36]
38         cmpeq   r1, r6
39         and     r2, r2, #1
40         eor     r2, r2, #1
41         sub     r2, r2, #1
42         it      ne
43         bicne   r2, r2, #2
44         tst     r2, #1
45         it      ne
46         movne   r5, #0
47         tst     r2, #2
48         it      ne
49         movne   r5, #1
50         tst     r2, #1
51         cmp.w   r10, #0
52         mov.w   r1, #0
53         str     r5, [sp, #8]
```

```
54          eor       r12, r5, #1
55          it        eq
56          moveq     r1, #1
57          tst.w     r1, r12
58          ite       eq
59          addeq.w r1, r10, #1
60          movne     r1, #0
61          cmp       r1, #0
62          mov.w     r6, #0
63          itttt     eq
64          mvneq     r6, #0
65          ldreq.w r2, [r4, r0, lsl #2]
66          ldreq     r4, [sp, #36]
67          cmpeq     r2, r4
68          and       r6, r6, #1
69          eor       r6, r6, #1
70          sub       r6, r6, #1
71          it        ge
72          bicge     r6, r6, #4096
73          tst       r6, #1
74          it        ne
75          movne.w r9, #0
76          tst       r6, #4096
77          it        ne
78          movne.w r9, #1
79          tst       r6, #1
80          cmp       r1, #0
81          mov.w     r6, #0
82          eor       r2, r9, #1
83          it        eq
84          moveq     r6, #1
85          tst       r6, r2
86          ite       eq
87          addeq     r1, #1
88          movne     r1, #0
89          cmp       r1, #0
90          itt       eq
91          subeq     r6, r0, #1
92          streq     r6, [sp, #28]
93          movs      r4, #0
94          cmp       r1, #0
95          it        ne
96          subne     r4, r1, #1
97          cmp       r4, #0
98          mov.w     r1, #0
99          mov       r5, r11
100         it        eq
101         moveq     r1, #1
102         tst       r1, r2
103         mov       r2, r7
104         itt       ne
105         ldrne     r2, [sp, #28]
106         movne     r5, r8
107         cmp       r4, #0
```

```
108         mov.w    r6, #0
109         it       eq
110         moveq    r6, #1
111         tst.w    r6, r9
112         ite      eq
113         addeq    r4, #1
114         movne    r4, #0
115         cmp      r4, #0
116         itt      eq
117         addeq    r6, r0, #1
118         streq    r6, [sp, #32]
119         movs     r6, #0
120         cmp      r4, #0
121         it       ne
122         subne    r6, r4, #1
123         cmp      r6, #0
124         mov.w    r4, #0
125         it       eq
126         moveq    r4, #1
127         tst.w    r4, r9
128         ittt     ne
129         movne    r2, r3
130         ldrne    r1, [sp, #32]
131         movne    r5, r1
132         str      r0, [sp, #16]
133         movs     r0, #0
134         cmp      r6, #0
135         it       ne
136         subne    r0, r6, #1
137         cmp      r0, #0
138         mov.w    r6, #0
139         mov      lr, r7
140         it       eq
141         moveq    r6, #1
142         tst.w    r6, r12
143         mov      r12, r11
144         itttt    ne
145         movne    r1, r5
146         strne    r1, [sp, #48]
147         movne    r1, r2
148         strne    r1, [sp, #44]
149         itt      ne
150         movne    r12, r5
151         movne    lr, r2
152         cmp      r0, #0
153         mov.w    r1, #0
154         it       eq
155         moveq    r1, #1
156         ldr      r2, [sp, #8]
157         tst      r1, r2
158         ite      eq
159         addeq    r0, #1
160         movne    r0, #0
161         mov.w    r10, #0
```

```
162         cmp      r0, #0
163         it       ne
164         subne.w  r10, r0, #1
165         cmp.w    r10, #0
166         mov.w    r0, #0
167         it       eq
168         moveq    r0, #1
169         tst      r0, r2
170         itttt    ne
171         movne    r0, #1
172         addne.w  r1, sp, #40
173         stmne.w  r1, {r0, r3, r8}
174         movne    r12, r11
175         it       ne
176         movne    lr, r7
177         ldr      r6, [sp, #4]
178         cmp.w    r10, #0
179         cmp.w    r10, #0
180         mov.w    r0, #0
181         it       eq
182         mvneq    r0, #0
183         ldr      r3, [sp, #24]
184         ldr      r7, [sp, #20]
185         itt      eq
186         ldreq    r1, [sp, #40]
187         cmpeq    r1, #0
188         and      r0, r0, #1
189         eor      r0, r0, #1
190         sub      r0, r0, #1
191         it       ne
192         bicne    r0, r0, #2
193         tst      r0, #1
194         it       ne
195         movne    r3, #0
196         tst      r0, #2
197         it       ne
198         movne    r3, #1
199         tst      r0, #1
200         cmp.w    r10, #0
201         mov.w    r0, #0
202         mvn.w    r1, r3
203         it       eq
204         moveq    r0, #1
205         tst      r0, r1
206         it       ne
207         movne    r7, #0
208         cmp.w    r10, #0
209         mov.w    r0, #0
210         mov.w    r1, #0
211         mov      r5, r2
212         it       eq
213         moveq    r0, #1
214         ands     r0, r3
215         itttt    ne
```

```
216         mvnne    r1, #0
217         ldrne    r2, [sp, #48]
218         ldrne    r4, [sp, #44]
219         cmpne    r4, r2
220         and      r1, r1, #1
221         eor      r1, r1, #1
222         sub      r1, r1, #1
223         it       lt
224         biclt    r1, r1, #2048
225         tst      r1, #1
226         it       ne
227         movne    r6, #0
228         tst      r1, #2048
229         it       ne
230         movne    r6, #1
231         tst      r1, #1
232         ldr      r4, [sp]
233         ldr      r2, [sp, #12]
234         cmp      r0, #1
235         it       eq
236         moveq    r7, r6
237         cmp.w    r10, #0
238         mov.w    r0, #0
239         str      r7, [sp, #20]
240         str      r3, [sp, #24]
241         mvn.w    r1, r7
242         it       eq
243         moveq    r0, #1
244         tst      r0, r1
245         it       ne
246         movne.w  r10, #1
247         ldr.w    r8, [sp, #48]
248         ldr      r3, [sp, #44]
249         ldr      r0, [sp, #16]
250         adds     r2, #1
251         mov      r11, r12
252         mov      r7, lr
253         bne.w    .LBB5_2
254         movs     r6, #0
255         cmp.w    r10, #0
256         it       ne
257         subne.w  r6, r10, #1
258         cmp      r6, #0
259         mov.w    r2, #0
260         it       eq
261         mvneq    r2, #0
262         ittt     eq
263         moveq    r1, #0
264         ldreq    r3, [sp, #40]
265         cmpeq    r3, #0
266         and      r2, r2, #1
267         eor      r2, r2, #1
268         sub      r2, r2, #1
269         it       ne
```

```
270        bicne    r2, r2, #2
271        tst      r2, #2
272        it       ne
273        movne    r1, #1
274        tst      r2, #1
275        cmp      r6, #0
276        mov.w    r3, #0
277        mov.w    r2, #0
278        mvn.w    r7, r1
279        it       eq
280        moveq    r3, #1
281        tst      r3, r7
282        cmp      r6, #0
283        it       eq
284        moveq    r2, #1
285        tst      r2, r1
286        it       ne
287        movne.w  r0, #-1
288        add      sp, #52
289        pop.w    {r4, r5, r6, r7, r8, r9, r10, r11, pc}
```

Listing A.7: Assembler-instructions of the binary-search example given in Listing A.4 when compiled with SP-transformation. Colors are used to categorize the instructions into Program- (black), Spill- (green), Move- (brown) and Guarding-instructions (blue).

List of Acronyms

AAPCS Procedure Call Standard for the ARM Architecture
ABI Application Binary Interface
APSR Application Program Status Register
ARM Acorn RISC Machines, later Advanced RISC Machines
CDG Control-Dependence Graph
CFG Control-Flow Graph
CPSR Current Program Status Register, deprecated synonym for APSR
CPU Central Processing Unit
DAG Directed acyclic graph
DFST Depth-First Spanning Tree
FCFG Forward Control Flow Graph
FPU Floating-Point Unit
FORTRAN The IBM Mathematical FORmula TRANslating System
GCC The GNU Compiler Collection
GSA Gated Single-Assignment
ID Input-Dependent
IFDS Interprocedural,Finite,Distributive,Subset [37]
IR LLVM Intermediate-Representation, The main language of LLVM
ISA Instruction Set Architecture
LCSSA Loop-Closed Static Single Assignment (form)
LLVM The LLVM-Project, formerly Low Level Virtual Machine. To reflect the extended scope of the LLVM-Project, nowadays LLVM is used as a name instead of an acronym
NID Non input-dependent
PDG Program Dependence Graph
PLL Phase-Locked Loop
PST Process Structure Tree
RA Register-Allocation
RISC Reduced Instruction Set Computer
RT Real-Time / Runtime
SCR Strongly Connected Region
SESE Single-Entry, Single-Exit
SP Single-Path
SSA Static Single Assignment (form)

TGSA Thinned Gated Single-Assignment [13]
UAL ARM Unified Assembler Language
WCET Worst-Case Execution Time

Bibliography

[1] F. E. Allen and J. Cocke. "A program data flow analysis procedure". In: *Communications of the ACM* 19 (3 Mar. 1976), pp. 137–. ISSN: 0001-0782. DOI: 10.1145/360018.360025. URL: http://doi.acm.org/10.1145/360018.360025.

[2] Frances E. Allen. "Control flow analysis". In: *SIGPLAN Not.* 5 (7 July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: http://doi.acm.org/10.1145/390013.808479.

[3] J. R. Allen et al. "Conversion of control dependence to data dependence". In: *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* Austin, Texas: ACM, 1983, pp. 177–189. ISBN: 0-89791-090-7. DOI: 10.1145/567067.567085.

[4] *ARM v7-M Architecture Reference Manual.* Tech. rep. ARM DDI 0403C_errata_v3. 2010.

[5] Larry Carter, Jeanne Ferrante, and Clark Thomborson. "Folklore confirmed: reducible flow graphs are exponentially larger". In: *SIGPLAN Not.* 38.1 (Jan. 2003), pp. 106–114. ISSN: 0362-1340. DOI: 10.1145/640128.604141. URL: http://doi.acm.org/10.1145/640128.604141.

[6] John Cocke. *Programming languages and their compilers: Preliminary notes.* Courant Institute of Mathematical Sciences, New York University, 1969. ISBN: B0007F4UOA.

[7] *Cortex-M3 Devices – Generic User Guide.* Tech. rep. ARM DUI 0552A (ID121610). 2010. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf.

[8] *Cortex-M3 – Technical Reference Manual.* Tech. rep. ARM DDI 0337I (ID072410). 2010. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337i/DDI0337I_cortexm3_r2p1_trm.pdf.

[9] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Trans. Program. Lang. Syst.* 13 (4 Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320.

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization". In: *ACM Trans. Program. Lang. Syst.* 9 (3 July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: http://doi.acm.org/10.1145/24039.24041.

[11] J. Gustafsson et al. “Input-Dependency Analysis for Hard Real-Time Software”. In: *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. The Ninth IEEE International Workshop on.* 2003, pp. 53–53. DOI: 10.1109/WORDS.2003.1267490.

[12] M. R. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop.* WWC ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.15. URL: http://dx.doi.org/10.1109/WWC.2001.15.

[13] Paul Havlak. “Construction of Thinned Gated Single-Assignment Form”. In: *In Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing.* Springer Verlag, 1993, pp. 477–499.

[14] Paul Havlak. “Interprocedural Symbolic Analysis”. PhD thesis. Center for Research on Parallel Computation, Rice University, May 1994.

[15] C. Healy et al. “Bounding loop iterations for timing analysis”. In: *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE.* 1998, pp. 12–21. DOI: 10.1109/RTTAS.1998.683183.

[16] M. S. Hecht and J. D. Ullman. “Characterizations of Reducible Flow Graphs”. In: *J. ACM* 21 (3 July 1974), pp. 367–375. ISSN: 0004-5411. DOI: 10.1145/321832.321835. URL: http://doi.acm.org/10.1145/321832.321835.

[17] Matthew S. Hecht and Jeffrey D. Ullman. “Analysis of a simple algorithm for global data flow problems”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* POPL ’73. Boston, Massachusetts: ACM, 1973, pp. 207–217. DOI: 10.1145/512927.512946. URL: http://doi.acm.org/10.1145/512927.512946.

[18] Matthew S. Hecht and Jeffrey D. Ullman. “Flow graph reducibility”. In: *Proceedings of the fourth annual ACM symposium on Theory of computing.* STOC ’72. Denver, Colorado, United States: ACM, 1972, pp. 238–250. DOI: 10.1145/800152.804919. URL: http://doi.acm.org/10.1145/800152.804919.

[19] Richard Johnson, David Pearson, and Keshav Pingali. “The program structure tree: computing control regions in linear time”. In: *SIGPLAN Not.* 29.6 (June 1994), pp. 171–185. ISSN: 0362-1340. DOI: 10.1145/773473.178258. URL: http://doi.acm.org/10.1145/773473.178258.

[20] Richard Johnson and Keshav Pingali. “Dependence-based program analysis”. In: *SIGPLAN Not.* 28.6 (June 1993), pp. 78–89. ISSN: 0362-1340. DOI: 10.1145/173262.155098. URL: http://doi.acm.org/10.1145/173262.155098.

[21] 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. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945. URL: http://doi.acm.org/10.1145/512927.512945.

[22] Raimund Kirner et al. “Beyond loop bounds: comparing annotation languages for worst-case execution time analysis”. English. In: *Software & Systems Modeling* 10.3 (2011), pp. 411–437. ISSN: 1619-1366. DOI: 10.1007/s10270-010-0161-0. URL: http://dx.doi.org/10.1007/s10270-010-0161-0.

[23] Donald E. Knuth. “An Empirical Study of FORTRAN Programs”. In: *Software - Practice and Experience* 1 (2 1971), pp. 105–133. DOI: 10.1002/spe.4380010203.

[24] D. J. Kuck et al. “Dependence graphs and compiler optimizations”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’81. Williamsburg, Virginia: ACM, 1981, pp. 207–218. ISBN: 0-89791-029-X. DOI: 10.1145/567532.567555. URL: http://doi.acm.org/10.1145/567532.567555.

[25] William Landi. “Undecidability of static analysis”. In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), pp. 323–337. ISSN: 1057-4514. DOI: 10.1145/161494.161501. URL: http://doi.acm.org/10.1145/161494.161501.

[26] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[27] Chris Lattner. “Macroscopic Data Structure Analysis and Optimization”. PhD thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, May 2005.

[28] E. J. McCluskey. “Minimization of Boolean functions”. In: *The Bell System Technical Journal* 35.5 (Nov. 1956), pp. 1417–1444.

[29] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. “Practical extensions to the IFDS algorithm”. In: *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*. CC’10/ETAPS’10. Paphos, Cyprus: Springer-Verlag, 2010, pp. 124–144. ISBN: 3-642-11969-7, 978-3-642-11969-9. DOI: 10.1007/978-3-642-11970-5_8. URL: http://dx.doi.org/10.1007/978-3-642-11970-5_8.

[30] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. “The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages”. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. PLDI ’90. White Plains, New York, United States: ACM, 1990, pp. 257–271. ISBN: 0-89791-364-7. DOI: 10.1145/93542.93578. URL: http://doi.acm.org/10.1145/93542.93578.

[31] Keshav Pingali and Gianfranco Bilardi. “Optimal control dependence computation and the Roman chariots problem”. In: *ACM Trans. Program. Lang. Syst.* 19.3 (May 1997), pp. 462–491. ISSN: 0164-0925. DOI: 10.1145/256167.256217. URL: http://doi.acm.org/10.1145/256167.256217.

[32] *Procedure Call Standard for the ARM Architecture*. Tech. rep. ARM IHI 0042E, current through ABI release 2.09. Nov. 30, 2012. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf.

[33] LLVM Project. *LLVM Language Reference Manual*. Aug. 2013. URL: http://www.llvm.org/docs/LangRef.html.

[34] Peter Puschner. “Experiments with WCET-Oriented Programming and the Single-Path Architecture”. In: *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. Feb. 2005, pp. 205–210.

[35] Peter Puschner and Alan Burns. “Writing temporally predictable code”. In: *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*. 2002, pp. 85–91. ISBN: 0-7695-1576-2.

[36] Peter Puschner et al. “Compiling for Time Predictability”. In: *Computer Safety, Reliability, and Security*. Ed. by Frank Ortmeier and Peter Daniel. Vol. 7613. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 382–391. ISBN: 978-3-642-33674-4. DOI: 10.1007/978-3-642-33675-1_35. URL: http://dx.doi.org/10.1007/978-3-642-33675-1_35.

[37] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *POPL ’95 Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, pp. 49–61.

[38] Thomas Reps, Mooly Sagiv, and Susan Horwitz. “Interprocedural DataflowAnalysis via Graph Reachability”. In: 1994, pp. 49–61.

[39] Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. “User-Input Dependence Analysis via Graph Reachability”. In: *Source Code Analysis and Manipulation, IEEE International Workshop on* (2008), pp. 25–34. DOI: 10.1109/SCAM.2008.22.

[40] *Stellaris® LM3S8962 Evaluation Board, User ’s Manual*. Tech. rep. EK-LM3S896 2-08. Feb. 9, 2010.

[41] *Stellaris® LM3S8962 Microcontroller DATA SHEET*. Tech. rep. DS-LM3S8962-7787. Sept. 4, 2010.

[42] Robert Tarjan. “Testing flow graph reducibility”. In: *Proceedings of the fifth annual ACM symposium on Theory of computing*. STOC ’73. Austin, Texas, USA: ACM, 1973, pp. 96–107. DOI: 10.1145/800125.804040. URL: http://doi.acm.org/10.1145/800125.804040.

[43] Alfred V. Aho ; Ravi Sethi ; Jeffrey D. Ullmann. *Compilerbau, 2. Auflage*. Oldenbourg, 1999.

[44] Sebastian Unger and Frank Mueller. “Handling irreducible loops: optimized node splitting versus DJ-graphs”. In: *ACM Trans. Program. Lang. Syst.* 24.4 (July 2002), pp. 299–333. ISSN: 0164-0925. DOI: 10.1145/567097.567098. URL: http://doi.acm.org/10.1145/567097.567098.

[45] W. Van Orman Quine. *The Problem of Simplifying Truth Functions.* Mathematical Association of America, 1952.

[46] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. "The Refined Process Structure Tree". In: *Business Process Management.* Ed. by Marlon Dumas, Manfred Reichert, and Ming-Chien Shan. Vol. 5240. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 100–115. ISBN: 978-3-540-85757-0. DOI: 10.1007/978-3-540-85758-7_10. URL: http://dx.doi.org/10.1007/978-3-540-85758-7_10.

[47] WG14. *ISO C Standard 1999, Committee Draft.* Tech. rep. ISO/IEC 9899:1999 draft. Sept. 2007. URL: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[48] Reinhard Wilhelm et al. "The worst-case execution-time problem–overview of methods and survey of tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: http://doi.acm.org/10.1145/1347375.1347389.